Titre: Title:	Fast, Scalable, and Flexible C++ Hardware Architectures for Network Data Plane Queuing and Traffic Management
Auteur: Author:	Imad Benacer
Date:	2019
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Benacer, I. (2019). Fast, Scalable, and Flexible C++ Hardware Architectures for Network Data Plane Queuing and Traffic Management [Thèse de doctorat, Polytechnique Montréal]. PolyPublie. https://publications.polymtl.ca/3812/

UNIVERSITÉ DE MONTRÉAL

FAST, SCALABLE, AND FLEXIBLE C++ HARDWARE ARCHITECTURES FOR NETWORK DATA PLANE QUEUING AND TRAFFIC MANAGEMENT

IMAD BENACER
DÉPARTEMENT DE GÉNIE INFORMATIQUE ET GÉNIE LOGICIEL
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

THÈSE PRÉSENTÉE EN VUE DE L'OBTENTION
DU DIPLÔME DE PHILOSOPHIÆ DOCTOR
(GÉNIE INFORMATIQUE)
AVRIL 2019

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Cette thèse intitulée:

FAST, SCALABLE, AND FLEXIBLE C++ HARDWARE ARCHITECTURES FOR NETWORK DATA PLANE QUEUING AND TRAFFIC MANAGEMENT

présentée par: BENACER Imad
en vue de l'obtention du diplôme de: Philosophiæ Doctor
a été dûment acceptée par le jury d'examen constitué de:

Mme NICOLESCU Gabriela, Doctorat, présidente
M. BOYER François-Raymond, Ph. D., membre et directeur de recherche
M. SAVARIA Yvon, Ph. D., membre et codirecteur de recherche
M. DAVID Jean Pierre, Ph. D., membre
M. DAHMANE Adel Omar, Ph. D., membre externe

DEDICATION

To my Parents
To my beloved Sister and Brothers

ACKNOWLEDGEMENTS

I would like to express my sincere and greatest gratitude to my supervisors, Dr. François-Raymond Boyer and Dr. Yvon Savaria. I am thankful for their constant support, encouragement and for their constructive advice and comments.

I would like to thank Normand Bélanger, researcher at Polytechnique Montréal for his suggestions and technical guidance.

Special thanks to Dr. Pierre Langlois for his help, support and guidance.

I am also thankful for my colleagues in the LASNEP, for the good times, mutual support and encouragement that we shared.

I would also like to thank the faculty, staff and students of the department who helped me to expand my knowledge and expertise. I would also like to thank them for every bit of help they provided, each in their own way.

My eternal gratitude to my family and friends for their unconditional encouragement and support, and to all those who have been there for me.

RÉSUMÉ

Les réseaux de communication actuels et ceux de la prochaine génération font face à des défis majeurs en matière de commutation et de routage de paquets dans le contexte des réseaux définis par logiciel (SDN–Software Defined Networking), de l'augmentation du trafic provenant des diffèrents utilisateurs et dispositifs connectés, et des exigences strictes de faible latence et haut débit. Avec la prochaine technologie de communication, la cinquième génération (5G) permettra des applications et services indispensables tels que l'automatisation des usines, les systèmes de transport intelligents, les réseaux d'énergie intelligents, la chirurgie à distance, la réalité virtuelle/augmentée, etc. Ces services et applications exigent des performances très strictes en matière de latence, de l'ordre de 1 ms, et des débits de données atteignant 1 Gb/s.

Tout le trafic Internet transitant dans le réseau est traité par le plan de données, aussi appelé traitement associé au chemin dit d'accès rapide. Le plan de données contient des dispositifs et équipements qui gèrent le transfert et le traitement des différents trafics. La hausse de la demande en bande passante Internet a accru le besoin de processeurs plus puissants, spécialement conçus pour le traitement rapide des paquets, à savoir les unités de traitement réseau ou les processeurs de réseau (NPU–Network Processing Units). Les NPU sont des dispositifs de réseau pouvant couvrir l'ensemble du modèle d'interconnexion de systèmes ouverts (OSI–Open Systems Interconnect) en raison de leurs capacités d'opération haute vitesse et de fonctionnalités traitant des millions de paquets par seconde. Compte tenu des besoins en matière de haut débit sur les réseaux actuels, les NPU doivent accélérer les fonctionnalités de traitement des paquets afin d'atteindre les requis des réseaux actuels et pour pouvoir supporter la nouvelle génération 5G. Les NPU fournissent divers ensembles de fonctionnalités, depuis l'analyse, la classification, la mise en file d'attente, la gestion du trafic et la mise en mémoire tampon du trafic réseau.

Dans cette thèse, nous nous intéressons principalement aux fonctionnalités de la mise en file d'attente et de gestion du trafic dans le contexte des équipements réseau à grande vitesse. Ces fonctionnalités sont essentielles et indispensables pour fournir et garantir la qualité de service (QoS–Quality of Service) pour divers trafics, en particulier dans le plan de données réseau des NPU, routeurs, commutateurs, etc., qui peuvent constituer un goulot d'étranglement car ils se trouvent dans le chemin critique. Nous présentons de nouvelles architectures pour les NPU et divers équipements réseau. Ces fonctionnalités sont intégrées en tant qu'accélérateurs externes qui peuvent être utilisés pour accélérer le fonctionnement via des FPGA (Field-Programmable Gate Arrays). Nous visons également à proposer un style de codage de haut niveau pouvant

être utilisé pour résoudre les problèmes d'optimisation liés aux différents besoins en réseau comme le traitement parallèle, le fonctionnement en pipeline et la réduction du temps de latence des accès mémoire, etc., permettant ainsi de réduire le temps de développement depuis la conception à haut niveau par rapport aux conceptions manuelles codées en langage de description matérielle (HDL–Hardware Description Language).

Pour la fonctionnalité de mise en file d'attente réseau, visant une solution à haut débit et à faible latence, nous proposons comme première idée une architecture matérielle de queue prioritaire (PQ–Priority Queue) en mode à une seule instruction plusieurs données en parallèle (SIMD–Single-Instruction–Multiple-Data) permettant de trier les paquets en temps réel, en prenant en charge indépendamment les trois opérations de base qui sont ajouter, retirer et remplacer des éléments en un seul cycle d'horloge. Notre objectif est de réduire la latence afin de prendre en charge au mieux les réseaux de la prochaine génération. La queue prioritaire implémentée est codée en C++. L'outil Vivado High-Level Synthesis (HLS) est utilisé pour générer une logique à transfert de registre (RTL–Register Transfer Logic) synthétisable à partir du modèle C++. Cette implémentation sur la carte FPGA ZC706 montre l'évolutivité de la solution proposée pour différentes profondeurs de queue, allant de 34 jusqu'à 1024 (1 Ki) éléments, avec des performances garanties. Elle offre une amélioration du débit de 10× atteignant 100 Gb/s par rapport aux travaux existant dans la littérature.

Comme deuxième idée, nous présentons une architecture de mise en file d'attente hybride destinée à la planification et la priorisation des paquets dans le plan de données du réseau. En raison de l'augmentation du trafic et des exigences strictes du réseau, une queue prioritaire à grande capacité, avec une latence constante et des performances garanties, est indispensable. Le système de mise en file d'attente prioritaire hybride proposé (HPQS–Hybrid Priority Queuing System) permet des opérations pipelinées avec une latence d'un seul cycle d'horloge. L'architecture proposée est entièrement codée en C++ et synthétisée avec l'outil Vivado HLS. Deux configurations sont proposées. La première configuration est destinée à la planification avec un système de files d'attente multiples pour laquelle nous montrons les résultats d'implémentation pour 64 à 512 queues indépendantes. La deuxième configuration est destinée à une queue prioritaire à grande capacité avec ½ million d'étiquettes de paquets pouvant être stockées. HPQS prend en charge les liaisons réseaux fonctionnant jusqu'à 40 Gb/s.

Pour la gestion du trafic réseau, nous proposons d'intégrer le gestionnaire de trafic (TM–Traffic Manager) en tant qu'accélérateur externe traitant uniquement les identificateurs des paquets. De cette manière, nous évitons de copier l'intégralité de ceux-ci, ce qui entraîne une plus grande efficacité d'implémentation (en termes de coût matériel et de performances) dans l'architecture système. En outre, la queue de priorité matérielle SIMD est utilisée pour trier

le temps de planification des paquets, ce qui est essentiel pour maintenir cette planification du trafic à de grande débits de liaison à 100 Gb/s. Dans cette solution, nous présentons une architecture programmable et évolutive, répondant aux besoins des équipements réseau, en particulier dans le contexte SDN. Ce TM est conçu pour faciliter le déploiement de nouvelles architectures via des plates-formes FPGA et pour rendre le plan de données programmable et évolutif. Ce TM prend également en charge l'intégration au plan de données programmable actuel avec le langage P4 (Programming Protocol-independent Packet Processors), en tant qu'une fonction C++ `extern`. Un réseau basé sur les flux permet de traiter le trafic en termes de flux plutôt que sur une simple agrégation de paquets individuels, ce qui simplifie la planification et l'allocation de bande passante pour chaque flux. La programmabilité en termes de création et de modification de bande passentes des divers flux apporte l'agilité, la flexibilité et l'adaptation rapide aux changements, permettant de répondre aux besoins du réseau en temps réel. Ce TM est capable de prendre en charge des paquets Ethernet de taille minimale de 84 octets, tout en planifiant les départs de paquets en deux cycles d'horloge.

ABSTRACT

Current and next generation networks are facing major challenges in packet switching and routing in the context of software defined networking (SDN), with a significant increase of traffic from different connected users and devices, and tight requirements of high-speed networking devices with high throughput and low latency. The network trend with the upcoming fifth generation communication technology (5G) is such that it would enable some desired applications and services such as factory automation, intelligent transportation systems, smart grid, health care remote surgery, virtual/augmented reality, etc. that require stringent performance of latency in the order of 1 ms and data rates as high as 1 Gb/s.

All traffic passing through the network is handled by the data plane, that is called the fast path processing. The data plane contains networking devices that handles the forwarding and processing of the different traffic. The request for higher Internet bandwidth has increased the need for more powerful processors, specifically designed for fast packet processing, namely the network processors or the network processing units (NPUs). NPUs are networking devices which can span the entire open systems interconnection (OSI) model due to their high-speed capabilities while processing millions of packets per second. With the high-speed requirement in today's networks, NPUs must accelerate packet processing functionalities to meet the required throughput and latency in today's networks, and to best support the upcoming next generation networks. NPUs provide various sets of functionalities, from parsing, classification, queuing, traffic management, and buffering of the network traffic.

In this thesis, we are mainly interested in the queuing and traffic management functionalities in the context of high-speed networking devices. These functionalities are essential to provide and guarantee quality of service (QoS) to various traffic, especially in the network data plane of NPUs, routers, switches, etc., and may represent a bottleneck as they reside in the critical path. We present new architecures for NPUs and networking equipment. These functionalities are integrated as external accelerators that can be used to speed up the operation through field-programmable gate arrays (FPGAs). Also, we aim to propose a high-level coding style that can be used to solve the optimization problems related to the different requirement in networking that are parallel processing, pipelined operation, minimizing memory access latency, etc., leading to faster time-to-market and lower development efforts from high-level design in comparison to hand-written hardware description language (HDL) coded designs.

For network queuing functionality, aiming for high throughput and low latency solution, we propose as a first idea a single-instruction–multiple-data (SIMD) hardware priority queue

(PQ) to sort out packets in real time, supporting independently the three basic operations of enqueuing, dequeuing, and replacing in a single clock cycle. We target to reduce latency to best support the upcoming next generation networks. The implemented PQ architecture is coded in C++. Vivado High-Level Synthesis (HLS) tool is used to generate synthesizable register transfer logic (RTL) from the C++ model. This implementation on the ZC706 FPGA shows the scalability of the proposed solution for various queue depths with guaranteed performance, it offers a 10× throughput improvement reaching 100 Gb/s when compared to prior works.

As a second idea, we present a fast hybrid priority queue architecture intended for scheduling and prioritizing packets in the network data plane. Due to increasing traffic and tight requirements of high-speed networking devices, a high capacity priority queue, with constant latency and guaranteed performance is needed. The proposed hybrid priority queuing system (HPQS) enables pipelined queue operations with one clock cycle latency. The proposed architecture is entirely coded in C++, and is synthesized with Vivado HLS tool. Two configurations are proposed. The first one is intended for scheduling with a multi-queuing system for which we report implementation results for 64 up to 512 independent queues. The second configuration is intended for large capacity priority queues with ½ million packet tags. The HPQS supports links operating up to 40 Gb/s.

For traffic management, we propose to integrate the traffic manager (TM) as a co-processor solution processing only packet tags. In this way, we avoid copying the entire packets which improves implementation efficiency (in terms of hardware cost, and performance) of the system architecture. Also, the SIMD PQ is used to sort out the scheduling time of packets, that is crucial to keep this traffic scheduling at gigabit link rates. In this solution, we present a programmable and scalable TM architecture, targeting requirements of high-speed networking devices, especially in the SDN context. This TM is intended to ease deployability of new architectures through FPGA platforms, and to make the data plane programmable and scalable. It supports also integration to today's programmable data planes with the popular P4 (Programming Protocol-independent Packet Processors) language, as a C++ `extern` function. Programmability brings agility, flexibility, and rapid adaptation to changes, allowing to meet network requirements in real time. Flow-based networking allows treating traffic in terms of flows rather than as a simple aggregation of individual packets, which simplifies scheduling and bandwidth allocation for each flow. This TM is capable of supporting links operating at 100 Gb/s, while scheduling packet departures in a constant 2-cycle per packet.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

5G	Fifth Generation Communication Technology
APQ	Augmented Priority Queue
ARM	Advanced RISC Machine
ASIC	Application-Specific Integrated Circuit
1 B	One Byte (1-B)
BRAM	Block RAM
CAM	Content Addressable Memory
CoS	Class of Service
DRAM	Dynamic RAM
DSP	Digital Signal Processor
DT	Decision Tree
DUT	Design Under Test
FF	Flip-Flop
FIFO	First-In First-Out
FPGA	Field-Programmable Gate Array
Gbps	Gigabit per second (Gb/s)
GPP	General Purpose Processor
GPU	Graphical Processing Unit
HDL	Hardware Description Language
HLD	High-Level Design
HLS	High-Level Synthesis
Hold	Dequeue–Enqueue (Replace) Operation
HPQS	Hybrid Priority Queuing System
II	Initiation Interval
IP	Internet Protocol
IPv4	Internet Protocol version 4
1 Ki	1024
LUT	Lookup Table
Mbps	Megabit per second (Mb/s)
Mpps	Million packets per second (Mp/s)
MP-SoC	MultiProcessor System-on-Chip
NA	Not Available/Not Applicable
NIC	Network Interface Card

ms	Millisecond
nm	Nanometer
ns	Nanosecond
NPU	Network Processing Unit
OPQ	Original Priority Queue
OSI	Open Systems Interconnection
P4	Programming Protocol-independent Packet Processors
PDI	Packet Descriptor Identification
PE	Processing Element
PIFO	Push-In First-Out
PQ	Priority Queue
P-Heap	Pipelined Heap
PHY	Physical Layer
QM	Queue Manager
QoS	Quality of Service
RAM	Random Access Memory
RISC	Reduced Instruction Set Computing
RR	Round-Robin
RTL	Register Transfer Logic
SDN	Software Defined Networking
SIMD	Single-Instruction–Multiple-Data
SLA	Service-Level Agreement
SoC	System-on-Chip
SRAM	Static RAM
TCAM	Ternary CAM
TM	Traffic Manager
VHDL	Very High Speed Integrated Circuit (VHSIC) HDL
Verilog	Verilog HDL
VLSI	Very Large Scale Integration
WFQ	Weighted Fair Queuing
WRED	Weighted Random Early Detection
WRR	Weighted Round-Robin
XML	Extensible Markup Language

LIST OF APPENDICES

CHAPTER 1 INTRODUCTION

1.1 Overview and Motivation

Over the recent few decades, Field-Programmable Gate Array (FPGA) technology and platforms have been a popular choice to implement computational intensive algorithms for networking, packet processing, real-time applications, etc. FPGAs are integrated silicon circuits, with a fine-grained reconfigurable processing architecture. Reconfigurability and a high degree of flexibility are the main advantages of FPGA-based designs. The trend to embed the FPGA (logic) side by side to processor-centric architectures in embedded systems started to grow due to the fact that General Purpose Processors (GPPs), like the Advanced RISC (Reduced Instruction Set Computing) Machine (ARM) and Xeon processors are not normally able to meet the high computational demands of the computationally intensive real-time applications in multimedia, communication, and signal processing domains. These applications can be accelerated by the integrated FPGA part of the modern System-on-Chips (SoCs) like in the Zynq-7 devices. At the other extreme, maximum feasible performance is offered by hardwired circuits widely known as Application-Specific Integrated Circuits (ASICs). However, ASICs generally offer the lowest flexibility in terms of programmability and reusability, while requiring larger development time.

FPGAs fill the gap between domain-specific processors such as Digital Signal Processors (DSPs), Graphical Processing Units (GPUs) and ASICs in the design space. In the past, the main idea of using FPGA was to design the entire architecture of a system based on a low-level Hardware Description Language (HDL). With the development of new tools such as Xilinx Vivado High-Level Synthesis (HLS), high-level languages such as C/C++ can be used to describe the system design for different applications and to target FPGA platforms.

Network Processing Units (NPUs) are networking devices which can span the entire Open Systems Interconnection (OSI) model due to their high-speed capabilities while processing millions of packets per second. With the high-speed requirement in today's networks, led by the growing demand to connect more devices to the Internet, NPUs must accelerate specific functionalities to meet the required throughput and latency in today networking traffic.

The aim of the proposed research is to develop new accelerators for NPUs and networking equipment like routers and switches. Also, we aim to propose a high-level coding style that can be used to solve the optimization problems related to the different requirements in networking, leading to faster time-to-market and lower development efforts from high-level

design in comparison to hand-written HDL coded designs.

Today's switches, routers, NPUs provide various sets of functionalities, from parsing, classification, scheduling and buffering of the network traffic. These functionalities can be supported by transformations applied to the traffic from the moment packets are received on input ports up to their transmission through destination ports. From the requirements of today's networks, switches must run at line rates to make local buffering manageable. For instance, 10 to 100 Gb/s are popular line rates. In this research, we are mainly interested in the queuing and traffic management functionalities. These functionalities are essential to provide and guarantee Quality of Service (QoS) to various traffic in the network data plane of NPUs, routers, switches, etc., and may represent a bottleneck as they reside in the critical path. One of the ideas of the proposed research is to integrate a Traffic Manager (TM) as a co-processor into the NPU. In this way, architectures that avoid copying the entire traffic (packets) can be proposed. This leads to better efficiency (in terms of hardware cost, and performance) of the system architecture. Also, priority queuing is used to sort out the scheduling time of packets. Therefore, a fast hardware Priority Queue (PQ) implementation is crucial to keep this scheduling scheme at gigabit link rates.

1.2 Problem Statement

Hardware implementation of dedicated circuits, accelerators, and co-processors is a possible answer to today's networking requirements. Designing such hardware is tedious, difficult and requires exhaustive testing and verifications with long development time, which usually end-up with an ASIC device. Addressing these problems in hybrid GPP-FPGA platforms or embedded systems forms the main goals of the proposed research.

HLS enables a smooth transition from a high-level coded design, usually in C/C++ and generates the corresponding Register Transfer Logic (RTL) in VHDL or Verilog. Moreover, the high-level testbench is used in the verification stage of the Design Under Test (DUT). The basic idea of this approach is to be able to design digital circuits at high-level abstraction, to target the required performances through directives and constraints to guide the HLS tool, abstracting away the details and the complexity of the design.

Co-processors for networking applications are surrounding most of the NPUs and the networking equipments. We aim to design co-processors or accelerators, from HLS to improve performance, hardware cost, while lowering the time needed for development in contrast to existing low-level designed co-processors. The proposed research aims to introduce and design efficient co-processors from HLS.

Hardware realization of the parallel processing designs is another influential factor in performance and hardware cost of the co-processor. For each Processing Element (PE), there are normally various possible implementation techniques with different requirements and target performance. The C/C++ language is used to describe sequential designs, while the same designs can run in parallel when mapped to hardware after HLS. Therefore, adequate high-level designs should be written in a way that allows efficient hardware implementation and rapid prototyping in FPGA platforms.

We target in this research the development of high-level design accelerators and their implementations. The complexity of these designs relies mostly on the intensive required calculations to meet specific performance criteria. Also, High-Level Design (HLD) is not straightforward, and many considerations should be taken into account especially the required resources and power consumption of the target FPGA platform.

1.3 Research Objectives

The main goal of this research is to propose and design high performance network data plane functions from high-level languages such as C/C++. HLD introduces two novelties: flexibility of the design, and capability to target multiple platforms as processors and FPGAs. The use of this technique in the design process allows efficient, rapid, effective and short time-to-market development as opposed to the low-level designs. Moreover, we target high throughput (10 Gbps and beyond) with very low latency, which is a must in the upcoming 5G communication technology and to meet today's networking requirements.

The following specific objectives are identified for this work. As part of this research, we proposed, designed, implemented and validated:

- A fast hardware PQ architecture from high-level code competing with low-level hand-written queuing systems.
- A high-capacity Hybrid Priority Queuing System (HPQS) with strict priority scheduling and priority queuing configurations providing guaranteed performance.
- A complete flow-based traffic manager architecture containing all the key functionalities from policing, scheduling, shaping and queue management targeting 10 – 100 Gb/s throughput.

Moreover, as a direct consequence of these research objectives, a high-level coding style was developed and explored to enable achieving the target performances such as enabling

parallel processing, maximizing the throughput, minimizing memory access latency, etc., and to validate the HLS design methodology. Also, the proposed designs target the most appropriate trade-offs among efficiency metrics, which includes performance, and hardware cost in this research. Yet, the final objective is to achieve the performance target of the low-level design for a given application, from high-level description language.

1.4 Research Contributions

The main objective of this research is to design and implement network data plane functions that are queuing and traffic management on hardware platforms with the aim to achieve high throughput, low latency with high-level design abstractions through HLS. This section reviews the contributions of the different parts of this thesis.

The main contributions achieved in priority queuing are:

- A fast register-based Single-Instruction–Multiple-Data (SIMD) PQ architecture, supporting the three basic operations of enqueuing, dequeuing and replacing in a single clock cycle regardless of the queue depth, while respecting the correct ordering of the PQ elements and queue invariants.
- A configurable FPGA-based PQ implementation obtained from HLS and providing easier implementation and more flexibility than low-level existing designs reported in the literature.
- Guaranteed performance (constant cycle queue operation) targeting 100 Gb/s throughput.

These contributions were published first in a conference paper entitled “A Fast Systolic Priority Queue Architecture for a Flow-Based Traffic Manager” in the IEEE International New Circuits and Systems Conference in 2016 [12], and in an extended paper version entitled “A Fast, Single-Instruction–Multiple-Data, Scalable Priority Queue” in the IEEE Transactions on Very Large Scale Integration (VLSI) Systems in 2018 [9].

The main contributions in high-capacity hybrid PQ are:

- Design of HPQS with two configurations. The first configuration (distinct-queues model) with partial and full sort capability supporting only en/dequeue operations for packet scheduling. The second configuration (single-queue model) supports a third queue operation (replace) for priority queuing.

- Analysis of HPQS operations leading to improvements that allowed matching the performance of hand-written RTL codes from a HLS design.
- Design space exploration under ZC706 FPGA and XCVU440 Virtex UltaScale device for resource (look-up tables and flip-flops), performance metrics (throughput, latency, and clock period), and power consumption analysis of the HPQS design.
- The performance are guaranteed, and are independent of the HPQS capacity. The HPQS throughput can reach 40 Gb/s for minimum 84 bytes Ethernet sized packets, with guaranteed 1 cycle per queue operation. HPQS can support half a million packet tags in a single FPGA.

These contributions were published first in a conference paper entitled "HPQ: A High Capacity Hybrid Priority Queue Architecture for High-Speed Network Switches" in the IEEE International New Circuits and Systems Conference in 2018 [13], and in an extended paper version entitled "HPQS: A Fast, High-Capacity, Hybrid Priority Queuing System for High-Speed Networking Devices" submitted to the IEEE Access in 2019 [10].

The main contributions to traffic management are:

- A flow-based TM architecture integrating core functionalities from policing, scheduling, shaping and queue management.
- An FPGA-prototyped TM architecture offering programmability, scalability, low-latency and guaranteed performance. This TM architecture exploits pipelined operations, and supports links operating beyond 40 Gb/s without loosing performance during flow updates (tuning), with minimum Ethernet sized packets.
- TM programmability can be supported with the popular P4 (programming protocol-independent packet processors) language, together with TM integration as a C++ `extern` function.

These contributions were published first in a conference paper entitled "A High-Speed Traffic Manager Architecture for Flow-Based Networking" in the IEEE International New Circuits and Systems Conference in 2017 [7], in a second conference paper entitled "Design of a Low Latency 40 Gb/s Flow-Based Traffic Manager Using High-Level Synthesis" in the IEEE International Symposium on Circuits and Systems in 2018 [8], and in an extended paper version entitled "A High-Speed, Scalable, and Programmable Traffic Manager Architecture for Flow-Based Networking" in the IEEE Access journal in 2019 [11].

1.5 Thesis Organization

This thesis is divided into 7 chapters. Chapter 2 reviews the important background material and related works that are used in this thesis. Chapter 3 describes queuing in the network data plane of NPUs, and the proposed hardware PQ architecture detailed in a paper entitled "A Fast, Single-Instruction–Multiple-Data, Scalable Priority Queue" [9]. Chapter 4 presents the HPQS detailed in a paper entitled "HPQS: A Fast, High-Capacity, Hybrid Priority Queuing System for High-Speed Networking Devices" [10]. Chapter 5 introduces and details the proposed flow-based traffic manager and its core functionalities in a paper entitled "A High-Speed, Scalable, and Programmable Traffic Manager Architecture for Flow-Based Networking" [11]. Chapter 6 presents a general discussion about the proposed implementations and highlights the limitations of our work. Chapter 7 concludes the thesis by summarizing our contributions and outlining our recommendations and future research directions.

CHAPTER 2 BACKGROUND AND LITERATURE REVIEW

In this chapter, we present a review of existing works in the literature related to our research subject. We start with an entry topic which is relevant to our research, that is the NPUs, where we survey the different existing processors, design methodologies, and network trends in this context. Then, we review two core functions for packet processing in NPUs, which are queuing and traffic management. Finally, we layout the HLS design methodology from high-level C/C++.

Accordingly, this chapter is organized as follows. Section 2.1 surveys the network processors. Section 2.2 presents significant relevant works found in the literature about queuing techniques with different design implementations, then we detail existing high capacity queuing systems. Section 2.3 discusses existing traffic managers, and Section 2.4 presents the HLS methodology and considerations for HLDs.

2.1 Network Architecture and Network Processors

The network is a set of connected devices and computers from homes, hotels, offices, etc. They are connected to the Internet through different means of communication, either by cables or wirelessly. The network architecture can be partitioned into three layers [18]:

- The application layer that contains the network applications such as security and network management, this layer assists the control layer in network architecture configuration.
- The control layer, also known as the slow path processing, that is essential for programming and managing the forwarding plane mainly with respect to routing.
- The data plane, also known as the fast path processing, that is responsible for forwarding the different network traffics.

The data plane contains networking devices that handle the forwarding and processing of the different traffics. These devices are: NPUs for packet processing, Network Interface Cards (NICs) connecting the different networking devices such as computers, servers, etc. Also, it contains the essential network components that are the routers and switches.

The network trend with the upcoming 5G communication technology is such that the number of connected people would be over 4 billion, using 10 to 100× more connected devices

from the different watches, sensors, smart cars, smart cameras, smart homes, etc. The performance requirements for latency is on the order of 1 ms, a total peak throughput rate of 10 Gbps and at least 100 Mbps connection rate whenever needed. These represent the networking requirements and trend numbers for the next few years [26, 63, 65]. The request for higher Internet bandwidth has increased the need for more powerful processors and NPUs, specifically designed for packet processing.

NPUs combine the flexibility of the GPPs with the increased performance of the ASICs [36]. NPUs are able to process millions of packets in core and access networks at wire speed. NPUs are used not only for the processing of the lower OSI layers such as layer 2 or 3, but also they are used for higher layers of the network (layer 4 to 7) such as the XML processors [23], and for deep packet inspection. The architectures of the NPUs vary from dataflow architectures such as the NPU from Xelerated [51] to multi-processors multi-threaded platforms such as the IXP2400 from Intel [35]. Many networking functions, that are extremely demanding, have been implemented in dedicated chips and are used as co-processors surrounding NPUs. Some examples of co-processors are traffic managers, network encryption processors, intrusion detection processors, content addressable memories (CAMs) for classification and Internet Protocol (IP) address lookup. FPGAs with embedded processors are ideal platform candidates for the development of NPUs accelerators that can achieve both flexibility and high performance.

2.1.1 Network Processor Categories

NPUs can be categorized according to their use, and the network requirements. The three main categories of NPUs [36] are entry level, mid-level and high-end:

2.1.1.1 Entry-level network processors

These NPUs process streams of packets of up to 1 to 2 Gbps, and are typically used for enterprise equipment. Applications for such access NPUs include telephony systems, cable modems, and optical networks.

2.1.1.2 Mid-level network processors

Targeting 2 to 5 Gbps processing throughput, these NPUs are used for service cards of communication equipment, data center equipment and layer 7 applications.

2.1.1.3 High-end network processors

Targeting 10 to 100 Gbps throughput, these NPUs are used mainly for core and metro networking, typically on the line cards of the equipment.

Figure 2.1 shows diversity for different NPUs with dissimilar architectures targeting throughput from 2.5 Gbps to beyond 40 Gbps spanning from OC-48 to OC-768 optical fiber carrier transmission standards or network links.

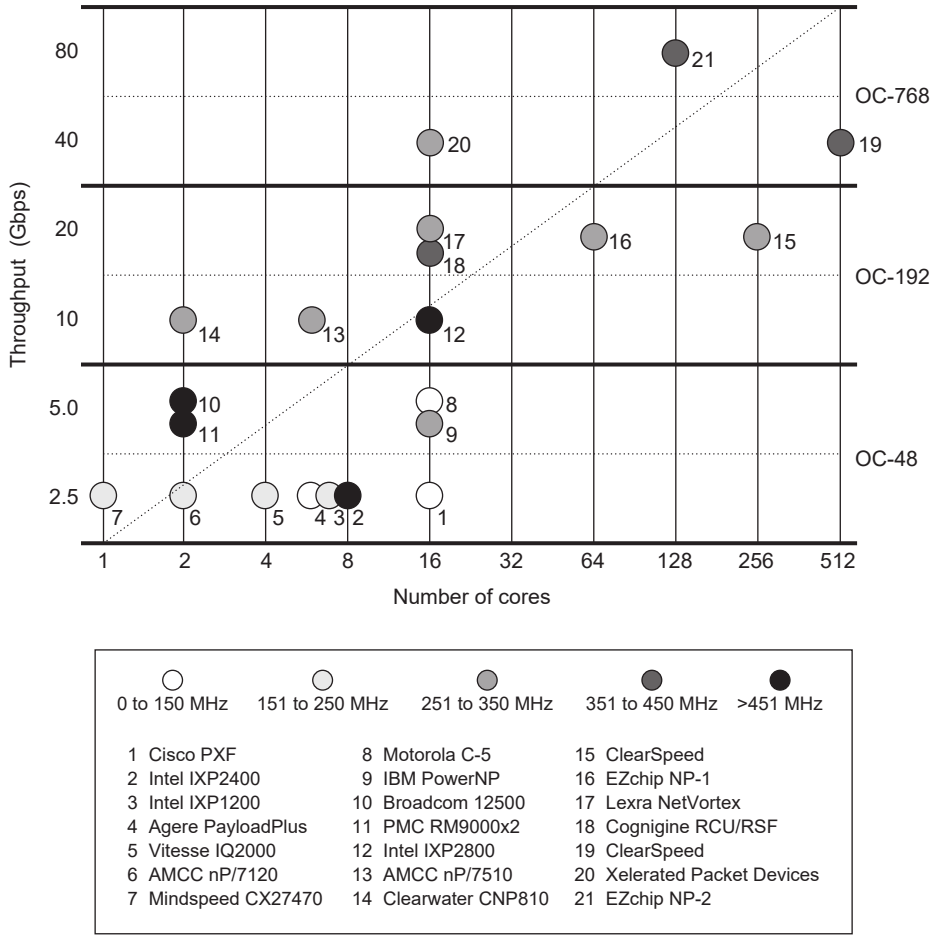


Figure 2.1 Diversity in the network processor performance [56], © 2002 IEEE.

2.1.2 Micro-Architectural Diversity

Another way to classify the network processors is by their specific topology of the PEs they contain. Figure 2.2 depicts the different topologies of PEs in network processors that can be found in parallel pools, pipeline, or mixed topology. The choice of a specific topology

rely mainly on the networking application. The pipeline and parallel configurations represent the two extremes: a pipeline configuration is generally used for high processing speed requirements in line cards at the lower layers of networking, whereas parallel configurations are used for networking applications of the higher layers (OSI layers 5 to 7). Moreover, there are various ways to increase the performance in NPUs; most, if not all, are achieved by using multiprocessing techniques and homogeneous/heterogeneous PEs. More details about the different architectures used in NPUs can be found in [36, 69].

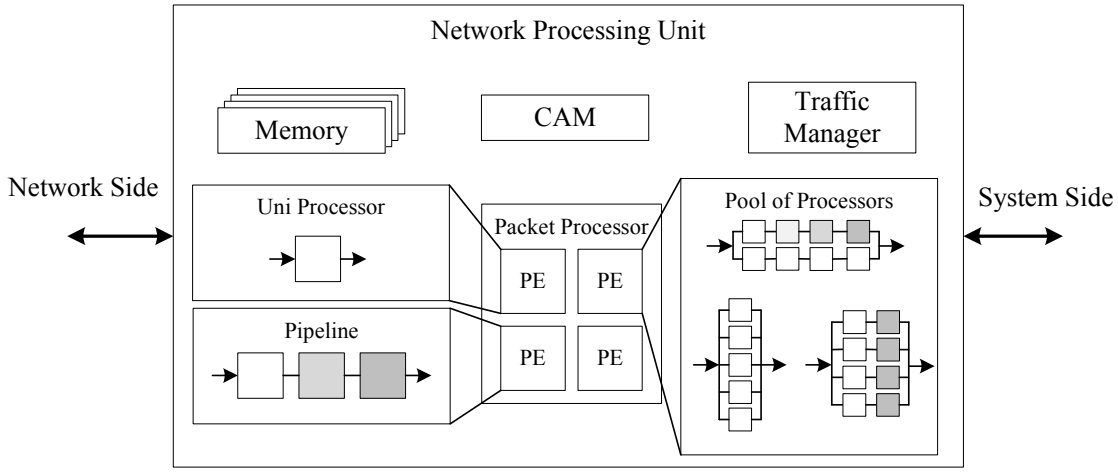


Figure 2.2 Processing elements topologies in a NPU.

2.1.3 Network Processor Functions

NPUs are specifically designed for packet processing targeting optimal performance of the networking functions in the OSI model. NPUs operations are subdivided into two main categories: control plane and data plane. The control plane deals with the slow path processing and usually handles non performance critical functions, i.e., route update in the routing table, statistics gathering, etc. mainly for system management. This is generally done with GPPs as the control plane requires little data parallelism by the host processing functions. The data plane deals with the fast path processing and usually is performance critical as it must operate at high-speed to meet the traffic requirement (traffic forwarding). NPUs are mostly optimized to implement data plane functionalities that require heavy data parallelism.

A general diagram for packet processing in a NPU is depicted in Figure 2.3. The packets enter from the ingress direction on the network side, and pass through the slow path using some kind of upper level processing, or pass the fast path through the NPU core functions such as framing, parsing/classification, etc. The packets are forwarded either to a switch fabric (system side) or to the network (line interface) again, in the egress direction. Our main

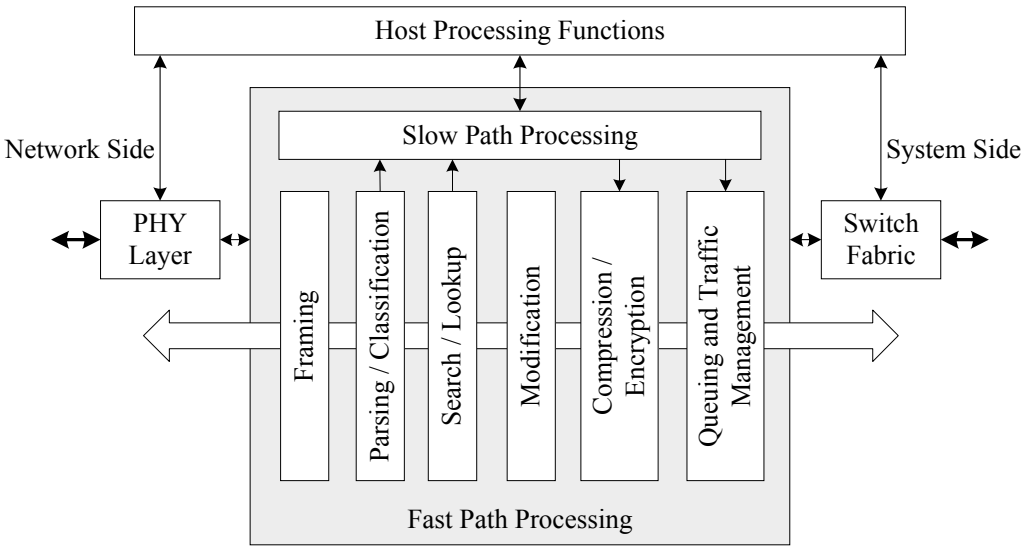


Figure 2.3 A general framework of packet processing based on [36].

focus is the queuing and traffic management, the last stage of packet processing in the NPU, where a decision should be taken on how to forward the packet on its way to destination. This process of forwarding a packet is accomplished according to different parameters related to the packet itself (i.e., priority), and to the system (for example QoS). This process is often complex and typically requires chained operations necessary for traffic management like policing, scheduling, shaping, etc., to meet the high-speed requirements.

In what follows, in Sections 2.2 and 2.3, we review works about queuing and traffic management core functionalities.

2.2 Queuing

In the literature, different queuing disciplines exist that implement the most common scheduling schemes used in the traffic managers and NPUs. They are subdivided mainly in four subcategories according to [36]: 1) First-In First-Out (FIFO), 2) Priority queuing, 3) Round-Robin (RR), and 4) Weighted fair queuing (WFQ). Our main focus in this research will be on the second subcategory namely the priority queuing and its various implementations.

PQs have been used for applications such as task scheduling [84], real-time sorting [58], event simulation [4, 24], etc. A PQ is an abstract data structure that allows insertion of new items and extraction of the high priority ones. In literature, different types of PQs have been proposed. Reported solutions span between the following: calendar queues [24], binary trees [57], shift registers [16, 27, 57], systolic arrays [48, 57], register-based arrays [39], and binary

heaps [15, 39, 41, 46]. However, a fair division can be made: PQs with fixed time operations or simply not depending on the PQ depth (number of nodes) against those with variable processing time. Moreover, PQs are also classified based on their implementations into software-, hardware-, or hybrid-based. Each class is further described in the next subsections.

2.2.1 Software Solutions

No software PQ implementation described in the literature can handle large PQs, with latency and throughput compatible with the requirements of today's high-speed networking devices. Existing software implementations are mostly based on heaps [39, 41, 76], with their inherent O(log(n)) complexity per operation, or alternatively O(s), where n is the number of keys or packets in the queue nodes, and s is the size of the keys (priority).

Research turned to the design of efficient high rate PQs through specialized hardware, such as ASICs and FPGAs. These PQs are reviewed in the following subsections.

2.2.2 Hardware Priority Queues

Moon [57] evaluated four scalable PQ architectures based on FIFOs, binary trees, shift registers, and systolic arrays. Moon showed that the shift register architecture suffers from a heavy bus loading problem as each new element has to be broadcasted to all blocks. This increases the hardware complexity and decreases the operating speed of the queue. The systolic array overcomes the problem of bus loading at the cost of resource usage higher than the shift register, needed for comparator logic and storage requirements. However, the systolic PQ does not fully sort in a single clock cycle, but still manages to enqueue and dequeue in a correct order and in constant time. On the other hand, the binary tree suffers from scaling problems including increased dequeue time and bus loading. The bus loading problem is due to the required distribution of new entries to each storage element in the storage block. The FIFO PQ uses one FIFO buffer per priority level. All such buffers are linked to a priority encoder to select the highest priority buffer. Compared to other designs, this architecture suffers from scaling the number of priority levels instead of the number of elements, requiring more FIFOs and a larger priority encoder. The total capacity investigated by Moon is 1024 (1 Ki) elements.

Brown [24] proposed the calendar queue, similar to the bucket sorting algorithm, operating on an array of lists that contains future events. It is designed to operate with O(1) average performance, but poorly performs with changing priority distribution. Also, extensive hardware support is required for larger priority values.

Sivaraman [72] proposed the Push-In First-Out (PIFO) queue. A PIFO is a PQ that allows elements to be enqueued into an arbitrary position according to the elements ranks (the scheduling order or time), while dequeued elements are always from the head. The sorting algorithm, called flow scheduler, enables O(1) performance. However, extensive hardware support is required due to the full ordering of the queue elements. The PIFO manages to enqueue, dequeue, and replace elements in the correct order and in constant time with a total capacity of 1 Ki elements.

2.2.3 Hybrid Priority Queues

The search for good options to implement efficient high rate large PQs led to exploring the use of hybrid PQs. Basically, they consist of hardware approaches, which are not scalable to the required queue sizes due to limited available resources, even though they produce the highest throughput, as described in detail in the previous subsection, with extension of the PQ operations to external storage using on-chip or off-chip memories.

Bhagwan [15] and Ioannou [41] proposed hybrid priority queue architectures based on a pipelined heap, i.e., a p-heap which is similar to a binary heap. However, the proposed priority queue supports en/dequeue operations in O(log(n)) time against a fixed time for the systolic array and shift register, where n is the number of keys. Also, these two implementations of pipelined PQs offer scalability and achieve high throughput, but at the cost of increased hardware complexity and performance degradation for larger priority values and queue sizes. The reported solutions implemented on ASICs had 64 Ki [41] and 128 Ki [15] as maximum queue capacities.

Kumar [46] proposed a hybrid priority queue architecture based on a p-heap implemented on FPGA supporting 8 Ki elements. This architecture can handle size overflow from the hardware queue to the off-chip memory. Moreover, Huang [39] proposed an improvement to the binary heap architecture. Huang's hybrid PQ combines the best of register-based array and BRAM-tree architectures. It offers a performance close to 1 cycle per replace (simultaneous dequeue-enqueue) operation. In this solution, the total implemented queue capacity is 8 Ki elements when targeting the ZC706 FPGA board.

Zhuang [87] proposed a hybrid PQ system exploiting an SRAM-DRAM-FIFO queue using an input heap, a creation heap and an output heap. The packet priorities are kept in sorted FIFOs called SFIFO queues that are sorted in decreasing order from head to tail. The three heaps are built with SRAMs, while the SFIFO queues extend the SRAM-based output heap to DRAM. Zhuang validated his proposal using a 0.13 µm technology under CACTI [67] targeting very large capacity and line rates: OC-768 and OC-3072 (40 and 160 Gb/s) while

the total expected packet buffering capacity reached 100 million packets.

Chandra [27] proposed an extension of the shift register based PQ of Moon [57] using a software binary heap. For larger queue capacity implementation up to 2 Ki, the resource consumption increases linearly, while the design frequency reduces logarithmically. This is a limitation for larger queues in terms of achieved performance and required hardware resources. Bloom [16] proposed an exception-based mechanism used to move the data to secondary storage (memory) when the hardware PQ overflows.

McLaughlin [53, 54] proposed a packet sorting circuit based on a lookup tree (a trie). This architecture is composed of three main parts: the tree that performs the lookup function with 8 Ki capacity, the translation table which connects the tree to the third part, the tag storage memory. It was implemented as an ASIC using the UMC 130-nm standard cell technology, and the reported PQ had a packet buffering capacity of up to 30 million packets tags.

Wang [77, 78] proposed a succinct priority index in SRAM that can efficiently maintain a real-time sorting of priorities, coupled with a DRAM-based implementation of large packet buffers targeting 40 Gb/s line rate. This complex architecture was not implemented, it was intended for high-performance network processing applications such as advanced per-flow scheduling with QoS guarantee.

Afek [1] proposed a PQ using TCAM/SRAM. This author showed the efficiency of the proposed solution and its advantages over other ASIC designs [53, 54, 87], but its overall rate degrades almost linearly with larger queue size while targeting 100 Gb/s line rate. Also, Afek presented an estimation of performance with no actual implementation.

Van [75] proposed a high throughput pipelined architecture for tag sorting targeting FPGA with 100 Gb/s line rate. This architecture is based on multi-bit tree and provides constant insert and delete operation requiring two clock cycles. The total supported number of packet tags is 8 Ki.

Table 2.1 summarizes some existing priority queues reported in the literature, their depths, the number of cycles between successive dequeue-enqueue (hold) operations, types, and the platform used for implementation.

2.3 Network Traffic Management

Traffic management deals essentially with packet handling and the way it should be forwarded. The core functionalities are policing, scheduling, shaping and queuing. The TM handles the scheduling of packets for transmission according to different parameters especially QoS. The TM also polices the packets, i.e. it allows them to proceed or to be discarded,

Table 2.1 Some existing priority queue architectures in the literature

Researcher	Architecture	Queue depth	# cycles per hold operation	Type	Priority level/ Platform
Moon (2000) [57]	Shift register	1 Ki	2 (no replace)	Hardware	16 / ASIC
	Systolic array	1 Ki	2 (no replace)	Hardware	16 / ASIC
Chandra (2010) [27]	Shift register	2 Ki	2 (no replace)	Hybrid	16 / FPGA (Cyclone II)
Huang (2014) [39]	Register/binary heap	8 Ki	1 : replace 11 : others	Hybrid	13 / FPGA (Zynq-7)
Ioannou (2007) [41]	Pipelined Heap	16 Ki	2 - 17 (no replace)	Hybrid	18 / ASIC
Sivaraman (2016) [72]	PIFO	1 Ki	2 : replace 4 : others	Hardware	16 / FPGA (Zynq-7)
McLaughlin (2008) [53]	Lookup tree (trie)	8 Ki	4: replace 8: others	Hybrid	NA / ASIC
Van (2015) [75]	Multi-bit tree	8 Ki	4 (no replace)	Hybrid	NA / FPGA (Virtex II)

and it shapes the transmitted stream to a specific traffic pattern. There exists two modes of operation of a TM around the NPU. In the flow-through mode, the TM is in the data path of the NPU. In the look-aside mode, the TM is outside the data path acting as a coprocessor.

Traditionally, traffic management has been implemented using hardwired state machines [73], then evolved from the NPUs [31, 35], to dedicated standalone solutions [20, 40], namely as coprocessors. Current solutions use dedicated traffic management integrated within the NPU to speed-up traffic processing while utilizing external storage memory for packet buffering proposes. There exist three different approaches for implementing the traffic management component of NPUs: software, hardware, or a hybrid of software and hardware [35]. Each of these methods has advantages for some functionalities and limitations in others targeting different design metrics: performance (throughput, delay), flexibility, scalability, cost, and QoS.

2.3.1 Software Implementation

In a software approach, all traffic management functions and algorithms are implemented through software programming. While a designer implements the algorithms and all associated modules, such as queues and tables in software, the NPU provides the required

synchronization and resource contention handling. Implementing traffic management in software is flexible by nature. However, the implementation is suitable only for low-speed traffic that does not stress excessively or exceed the software implementation capabilities. Traffic management requires the update of thousands of individual packet states, flows information, tables, etc. However, the sequential nature of standard software implementations results in poor performance. Consequently, traffic management can be effectively implemented in hardware since parallel and high-performance capabilities can be achieved, while supporting the networking requirements.

2.3.2 Hardware Implementation

Typically, this solution targets high-speed requirements through the design of traffic management functionalities directly into hardware. As high data rates are very challenging to achieve, hardware can be the only solution. This solution is most suitable for algorithms that undergo minimal changes (less flexible). The designer must customize the design and enable programmability of the traffic management functions at runtime such as queuing and scheduling. An example of such a TM is The Agere APP750TM network processor chipset [35].

2.3.3 Software/Hardware Implementation

Hybrid solutions allow software programmability and flexibility while maintaining the performance advantage of the hardware solution. This type of architecture implements key modules (most compute intensive as the core functionalities of policing, scheduling, shaping, and queuing) in hardware, while the software determines the strategy to be applied. Examples of such TMs are the EZchip NP-5 [31], and the Agere APP500 family of NPUs with integrated traffic management [35].

2.3.4 Traffic Managers Review

Available commercial TM solutions are either rigid because they rely on ASIC implementations, or require high-speed processing platforms [3, 21, 40, 80] and usually are considered as independent processing elements attached to a flexible pipeline in the NPU. In the research community, especially academia, only a few works have been published about a complete TM architecture [60, 86], others only focus on specific functionalities implemented in ASIC such as scheduling [28], or in FPGAs such as congestion management [34] and scheduling [44, 53, 54].

Paulin [64] proposed a MultiProcessor System-on-Chip (MP-SoC) architecture for traffic management of IPv4 forwarding. The proposed platform is composed of multiple configurable hardware multi-threaded processors, with each processor running part of the traffic management features or tasks. To process more traffic and to cope with network requirements, this architecture requires more processors, eventually limiting its scalability.

Zhang [86] proposed a complete TM implemented in an FPGA platform, focusing on the programmability and scalability of the architecture to address today's networking requirements. However, the queue management solution that was adopted slows down the entire system with at least 9 cycles per enqueue/dequeue operation, and an implementation running at 133 MHz. This TM solution achieved around 8 Gb/s for minimum size 64 byte packets.

Khan [42] proposed a traffic management solution implemented with dedicated circuits that can support 5 Gb/s with full duplex capabilities. Khan showed all the design steps up to the physical realization of a TM circuit. This solution remains rigid as it targets an ASIC. This design choice limits its ability to support future networking needs.

Table 2.2 summarizes the TM solutions offered by commercial vendors and published by academia, along with the platform for which they were developed, their configuration and the reported throughput.

2.4 High-Level Synthesis Design

As mentioned in the research objective Section 1.3, the proposed research aims to design, validate, and improve the efficiency of various circuits designed at high-level and targeting embedded systems, especially FPGA platforms, multi-core processing platforms like Xeon/FPGA [38], and Xilinx Zynq-7. Moreover, as described in Section 2.1, NPU packet processing functions requirements are demanding, and need extensive design space exploration, especially when coding at high-level with the intent of obtaining effective low-level designs. During recent years, specific high-level development tools were introduced. They employ particular methodologies to automate this exploration with at least 5 to 10× gain in productivity [55, 74]. However, special care should be taken during design steps. For instance, it is not always clear what low-level design will result from a high-level specification. Also, many aspects need special consideration such as the type of interface, the target resources and resulting performances (throughput and latency), etc.

Table 2.2 Some existing traffic management solutions in the literature

Company / Researcher	Platform	Configuration	Throughput (Gb/s)
Zhang (2012) [86]	FPGA (Virtex-5)	Look-aside	8
Altera (2005) [40]	FPGA (Stratix II)	Flow-through	10
Broadcom (2012) [21]	ASIC	Flow-through	Up to 200
Mellanox (Ezchip) (2015) [32]	ASIC	—	Up to 400
Agere (2002) [2]	ASIC	Flow-through	10
Xilinx (2006) [79]	FPGA (Virtex-4)	Look-aside	—
Paulin (2006) [64]	MP-SoC (90 nm)	Look-aside	2.5
Khan (2003) [42]	ASIC (150 nm)	Look-aside	5
Bay (2007) [3]	ASIC (110 nm)	Flow-through	50 or 12×4

2.4.1 High-Level Design

One of the major challenges in today's design is to reduce the development cycle and lower the time-to-market. The two main concepts behind design acceleration strategies are design reuse and rising of the abstraction level. Our main focus is on the latter, by exploring the Xilinx Vivado HLS tool [82]. This tool enables the synthesis of hardware circuits directly from a high-level description developed in C, C++, or SystemC. In this work we aim to code with the highest supported standards like C++0x (as the tool does not currently fully support C++11 or later). The main advantages of this tool are the automation of the entire design flow from a HLD into RTL equivalent as VHDL, Verilog, and SystemC, with no need to manually hand code the low-level RTL design as the HLS tool carries out the whole task.

The main aspect of HLS is that the designed functionality and its hardware implementation are kept separate. The high-level based description does not implicitly fix the hardware architecture and it gives more options for design space exploration with higher flexibility. However, this is not true for direct RTL-level design as it targets one specific architecture at a time. In FPGA designs, the considered layers of abstractions are usually categorized as structural, RTL, behavioral and high-level for any design flow, as shown in Figure 2.4. The lowest level

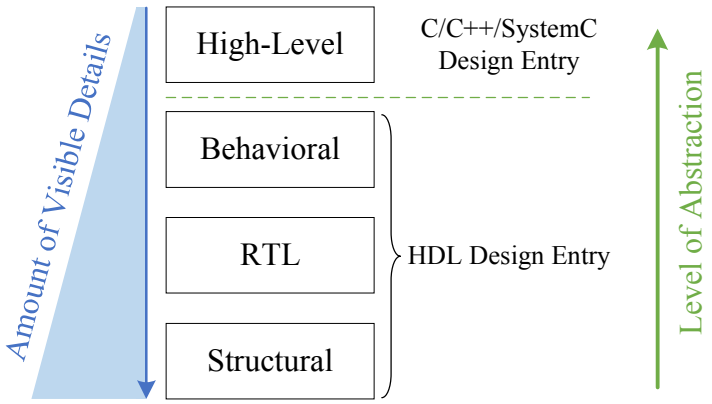


Figure 2.4 Abstraction levels in FPGA designs based on [29].

of abstraction requires instantiating the basic blocks of the target FPGA platform consisting of Look-Up Tables (LUTs) and Flip-Flops (FFs), BRAMs, and input/output requirements, with the maximum visibility of detail for place and routing (this can be done manually or automated by a synthesizer). The well-known and most commonly used abstraction level is RTL, where the designer describes the operation occurring between registers and logic, and many synthesis tools are designed to work with this level of abstraction. The behavioral description deals with describing the design in a more algorithmic way not as register to register communication in contrast to RTL. The last and higher layer of abstraction uses high-level description languages for system design modeling in a more algorithmic level of abstraction, which hides away the details of the hardware, focusing only on the algorithmic functionality of the coded design.

Our motivation for using HLS design is that we can use a high-level representation abstracting low-level details, from which the description of the entire design becomes much easier and simpler by coding the design in C/C++. Moreover, the designer directs the HLS process from which the Vivado HLS tool implements the details. A general design flow with HLS is depicted in Figure 2.5. The most important design aspects in HLS is that the separation of functionality and implementation implied in HLS means that the source code does not fix the architecture. Also, HLS offer several valuable options like functional verification and RTL co-simulation with the same high-level coded testbench, which leads to much faster design than using directly a HDL design methodology in VHDL/Verilog. Additionally, HLS provides capabilities to create and modify the architecture by applying directives and constraints, in contrast to rewrite the source code entirely in case of RTL designs. More details are given in the next subsection.

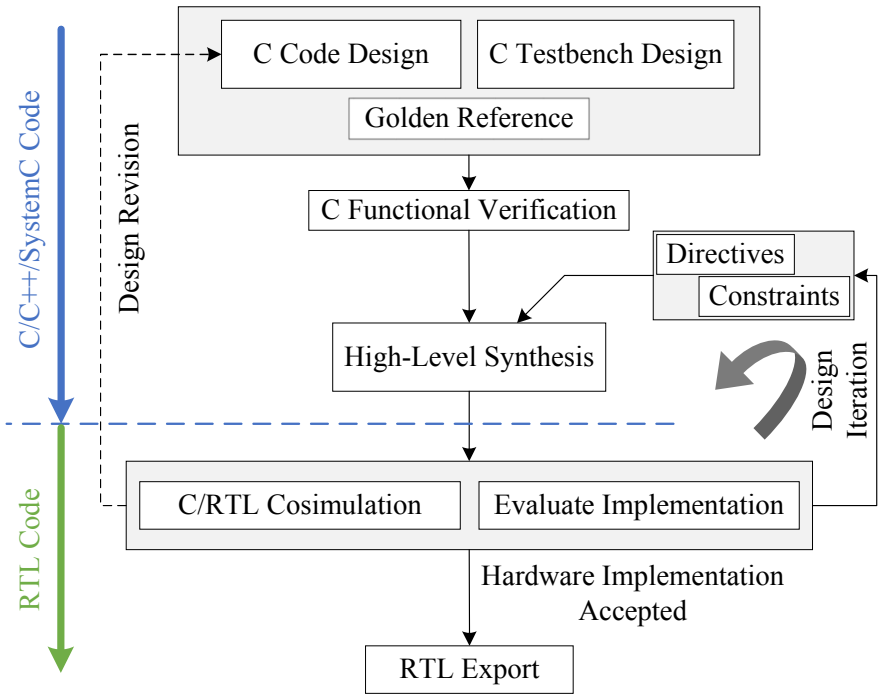


Figure 2.5 An overview of the Vivado HLS design flow based on [29].

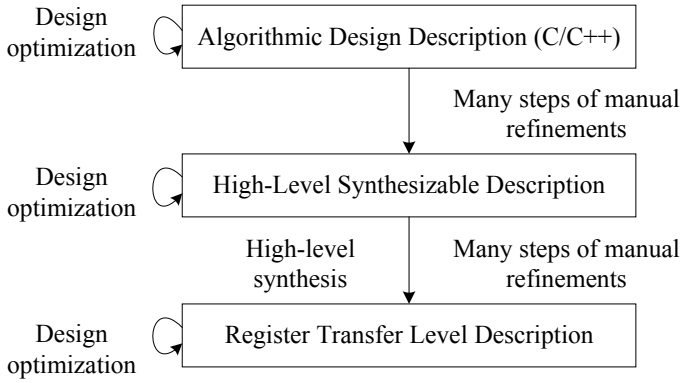


Figure 2.6 High-level design flow optimization based on [52].

2.4.2 HLS Design Optimization

Using an HLS design flow is not straightforward. The first step relies on the designer trusting the HLS tool to implement the design correctly and efficiently in RTL. The second step is to understand how to guide the tool over this process. Figure 2.6 illustrates a high-level design flow that starts from high-level design description, appropriate design iterations for refinement (code optimization and enhancement), verification and testing are considered to eliminate bugs, errors, etc., as the first step in any HLS design. Then, design implementation metrics

should be defined such as the target resource usage, achieved throughput, clock period, design latency, input/output requirements, etc., which are closely related to the design process and form part of it. These metrics can be controlled through directives and constraints applied during the HLS process [83].

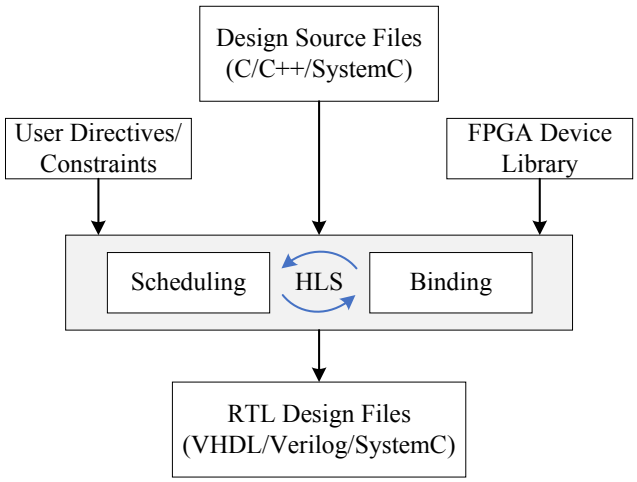


Figure 2.7 Vivado HLS scheduling and binding processes based on [29].

The HLS process is initiated by specifying the design files. The designer must specify a target device and appropriate directives / constraints he wants to apply. The Vivado HLS process as depicted in Figure 2.7 can be described in three steps:

- Extraction of data and control paths from the high-level design files.
- Scheduling and binding of the RTL in the hardware targeting specific device library.
- Optimizations dictated by the designer to guide the HLS process during scheduling and binding through:
    - Directives: There are various types of directives available which map to certain features of the code, enabling the designer to dictate, for example, how the HLS tool treats loops, arrays, input/output interface, identified in the high-level code, the latency of a particular piece of code, etc. Consequently, with knowledge of the available directives, the designer can optimize according to the defined design metrics and target the required design performance.
    - Constraints: The designer can specify and bound some features of the design in a global way, for example the clock period, resource utilization, memory partition, etc. This simplifies greatly the optimization of the design, and makes it easy to validate the implementation conformity to the defined metrics.

The user directives and constraints are the only way to guide the HLS tool to meet the requirements, besides the design files. The RTL generated code can be further optimized by specific tools like hardware synthesizers, but once the hardware implementation is accepted and the RTL is exported from the tool, the designer has a complete view of his design and achieved performance.

2.4.3 High-Level Accelerators Design

The proposed high-level coded designs are evaluated, refined through HLS design methodology until the desired performance metrics are achieved. Generally, the final product represents two designs. The first one is expressed at high-level in C/C++ and the other one is expressed at low-level in VHDL/Verilog. The two designs are similar in their application, but only target different platforms, the former targets a GPP (a general purpose processor software solution), while the latter targets an embedded system like an FPGA platform (a hardware solution).

The HLS design methodology requires by essence a high-level coded design, and by definition, the use of high-level is to abstract away the low-level details of interconnection, time synchronization, etc., while the entire focus is on the application itself. Moreover, the HLS design methodology has been widely studied since its introduction in the early 1990s. Many tools like Vivado HLS do exist with automation capability to convert a high-level description of an application to a low-level RTL model without any human intervention except through directives and constraints. One of our most important objectives is to propose HLD architectures and accelerators that meet the target performance metrics of resource usage, timing and throughput, while competing with low-level coded designs.

The most important aspect from HLD is to achieve the required performance at minimum design effort. Thus it allows faster design space exploration through the different optimization options. Once a functional high-level code is written according to the guide (basic knowledge to know about the hardware capability and limitations) provided by the HLS tool [83], the most important challenge is to address the optimization problem and try to find the best technique (directives and constraints) that should be used to achieve the target performance and design metrics.

CHAPTER 3 ARTICLE 1: A FAST, SINGLE-INSTRUCTION–MULTIPLE-DATA, SCALABLE PRIORITY QUEUE

This chapter contains the paper entitled "A Fast, Single-Instruction–Multiple-Data, Scalable Priority Queue" published in the IEEE Transactions on Very Large Scale Integration (VLSI) Systems[1]. We started our work by analyzing the context of network data plane queue management in NPUs and their implementations in order to identify opportunities for acceleration and efficient hardware implementation that offer guaranteed performance. We thus proposed a fast hardware PQ based on a SIMD architecture. The hardware PQ supports the three basic operations of enqueuing, dequeuing, and replacing. Our solution was completely implemented under the ZC706 FPGA (Zynq-7), using only logic resources. This implementation was obtained from HLS. Our results show improvements over the handwritten-HDL implementations from the literature.

Note that for those who read chapter 2, Sections 2.2 and 2.3, can skip Section 3.2 of this chapter.

Abstract—In this paper, we address a key challenge in designing flow-based traffic managers (TMs) for next-generation networks. One key functionality of a TM is to schedule the departure of packets on egress ports. This scheduling ensures that packets are sent in a way that meets the allowed bandwidth quotas for each flow. A TM handles policing, shaping, scheduling, and queuing. The latter is a core function in traffic management and is a bottleneck in the context of high-speed network devices. Aiming at high throughput and low latency, we propose a single-instruction–multiple-data (SIMD) hardware priority queue (PQ) to sort out packets in real time, supporting independently the three basic operations of enqueuing, dequeuing, and replacing in a single clock cycle. A proof of validity of the proposed hardware PQ data structure is presented. The implemented PQ architecture is coded in C++. Vivado high-level synthesis is used to generate synthesizable register transfer logic from the C++ model. This implementation on a ZC706 field-programmable gate array (FPGA) shows the scalability of the proposed solution for various queue depths with almost constant performance. It offers a 10× throughput improvement when compared to prior works, and it supports links operating at 100 Gb/s.

[1]I. Benacer, F.-R. Boyer, and Y. Savaria, "A Fast, Single-Instruction–Multiple-Data, Scalable Priority Queue," in IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 26, no. 10, pp. 1939-1952, Oct. 2018, © 2018 IEEE. This work was supported in part by the Natural Sciences and Engineering Research Council of Canada (NSERC), in part by Prompt Québec, and in part by Ericsson Research Canada.

3.1 Introduction

With the increasing number of Internet and mobile service subscribers, demand for high-speed data rates and advanced applications such as video sharing and streaming is growing fast. There is an ongoing process to define the next-generation communication infrastructure (5G) to cope with this demand. Yet, it is obvious that very low-latency packet switching and routing will be a major challenge to support life critical systems and real-time applications in the 5G context [62].

In network processing units (NPUs), packets are normally processed at wire speed through different modules. For example, a traffic manager (TM) provides queuing and scheduling functionalities [60, 86]; this is quite demanding because packet scheduling priorities are implicit and depend on several factors (protocols, traffic, congestion, etc.). One of the feasible solutions is to tag related packets with flow numbers [60] as soon as they enter the network. This helps allocating bandwidth and simplifies scheduling by alleviating the processing of individual packets in terms of flows or simply groups of packets.

With the current thrust toward software-defined networking [85], it becomes natural to associate each group of packets to a flow. For example, in cellular networks, bandwidth is assigned to subscribers, so each packet is already part of a flow with some bandwidth assigned to it. Thus, this flow tagging could become part of the context of the next-generation networking equipment.

Real-time applications, such as video streaming, require quality-of-service (QoS) guarantees such as average throughput, end-to-end delay, and jitter. To provide QoS guarantees, network resource prioritization matching requirements must be achieved by assigning priorities to packets according to the corresponding incoming flow information, which can represent specific types of applications, services, etc. To implement this priority-based scheduling, priority queues (PQs) implemented in hardware have been used to maintain real-time sorting of queue elements at link speeds. Hence, a fast hardware priority queue is crucial in high-speed networking devices (more details are given in Section 3.2).

The PQs have been used for applications such as task scheduling [84], real-time sorting [58], and event simulation [4, 24]. A PQ is an abstract data structure that allows insertion of new items and extraction in priority order. In the literature, different types of PQs have been proposed. Reported solutions span between the following: calendar queues [24], binary trees [57], shift registers [16, 27, 57], systolic arrays [48, 57], register-based arrays [39], and binary heaps [15, 39, 41, 45, 46]. Existing solutions can be partitioned in two classes: PQs with fixed time operations or processing time that do not depend on the queue depth (number of

nodes) and those with variable processing time.

This paper presents the following contributions.

1. A fast register-based single-instruction–multiple-data (SIMD) PQ architecture, supporting the three basic operations of enqueuing, dequeuing, and replacing. Our novel approach has modified the sorting operation in a way to restore required invariants in a single clock cycle for the entire queue. Also, we provide a detailed proof of the correctness of the PQ with respect to various desired properties. It will be shown that after placement and routing, the required clock period is almost constant regardless of the queue depth.

2. A configurable field-programmable gate array (FPGA)-based PQ implementation using high-level synthesis (HLS), entirely coded in C++ to facilitate implementation (by raising the level of abstraction) and provide more flexibility with faster design space exploration than other works seen in the literature, which use low-level coding in Verilog, VHDL, etc. [41, 48, 57, 60, 86].

3. A queuing management system capable of providing at least 103 Gb/s for 64-B sized packets (see Section 3.6). Also, a fixed, stable throughput, independent of the queue depth, is achieved as compared to other architectures [39, 41, 45, 46, 57, 86].

The remainder of this paper is organized as follows. In Section 3.2, we present a literature review of some existing traffic management and PQ implementations. In Section 3.3, we describe the architecture of a generic TM with its underlying modules. In Section 3.4, we present our hardware PQ with the proof of its validity, and a tradeoff analysis in terms of space versus time complexities. In Section 3.5, we present the HLS methodology, and the various explored directives/constraints to target desired resource usage and performance. In Section 3.6, hardware implementations of the proposed design with comparisons to existing works in the literature are discussed, and Section 3.7 draws conclusions.

3.2 Related Work

In this section, we present different traffic management works and solutions seen in the literature, and then, we detail well-known priority queuing models and their expected performances.

3.2.1 Traffic Management Solutions

Traffic management implementation evolved from network processor units [2, 21, 32] to dedicate stand-alone solutions [40, 79], namely, as coprocessors. Current solutions use dedicated traffic management integrated within NPUs to speed-up traffic processing, with external memories for packet buffering and queuing purposes.

The available traffic management solutions in the literature are essentially commercial products [2, 21, 32, 40, 79], which are usually closed. Few works about traffic management were published by academia. Zhang *et al.* [86] proposed a complete TM architecture implemented in an FPGA platform. Zhang focused on the programmability and scalability of the architecture in relation to today's networking requirements. However, the queue manager (QM) slows down the entire system with at least 9 cycles per action to enqueue/dequeue a packet, while running at 133 MHz. This TM achieved around 8 Gb/s for minimum size packets of 64 B. Khan *et al.* [42] proposed a traffic management solution implemented with dedicated circuits, supporting 5 Gb/s with full-duplex capabilities. Khan showed all the design steps up to the physical realization of a TM circuit. As Khan opted for an application-specified integrated circuit (ASIC), the reported solution remains rigid and has limited applicability for supporting future networking needs, such as increasing traffic demand and link speeds.

Table 3.1 summarizes the TM solutions offered by commercial vendors and published by academia, along with the platform for which they were developed, and their maximum achievable throughput.

Table 3.1 Traffic management solutions

Company / Researcher	**Platform**	**Throughput (*Gb/s*)**
Zhang (2012) [86]	FPGA Virtex-5 (65 nm)	8
Agere (2002) [2]	ASIC	10
Broadcom (2012) [21]	ASIC	Up to 200
Mellanox (Ezchip) (2015) [32]	ASIC	Up to 400
Xilinx (2006) [79]	FPGA Virtex-4 (90 nm)	NA
Altera (2005) [40]	FPGA Stratix II (90 nm)	10
Khan (2003) [42]	ASIC (150 nm)	5
Bay (2007) [3]	ASIC (110 nm)	Up to 50

3.2.2 Priority Queues

Previous reported PQs can be classified as software- or hardware-based. Each class is further described in Sections 3.2.2.1 and 3.2.2.2.

3.2.2.1 Software Solutions

No software PQ implementation in the literature can handle large PQs, with latency and throughput compatible with the requirements of today's high-speed networking systems. Existing software implementations are mostly based on heaps [39, 41, 76], with their inherent O(log(n)) complexity per operation, or alternatively O(s), where n is the number of keys or packets in the queue nodes, and s is the size of the keys (priority).

Research turned to the design of efficient high rate, and large PQs obtained by the use of specialized hardware, such as ASICs and FPGAs. These PQs are reviewed in Section 3.2.2.2.

3.2.2.2 Hardware Priority Queues

Moon *et al.* [57] evaluated four scalable PQ architectures based on: (first-in first-outs) FIFOs, binary trees, shift registers, and systolic arrays. This author showed that the shift register architecture suffers from a heavy bus loading problem as each new element has to be broadcasted to all blocks. This increases the hardware complexity and decreases the operating speed of the queue. The systolic array overcomes the problem of bus loading at the cost of higher resource usage than the shift register, needed for comparator logic and storage requirements. Similar to our design, the systolic PQ does not fully sort in a single clock cycle, but still manages to enqueue and dequeue in a correct order and in constant time. On the other hand, the binary tree suffers from scaling problems including increased dequeue time and bus loading. The bus loading problem is due to the required distribution of new entries to each storage element in the storage block.

The FIFO PQ architecture described by Moon *et al.* [57] uses one FIFO buffer per priority level. All such buffers are linked to a priority encoder to select the highest priority buffer. Compared to other designs, this architecture suffers from scaling the number of priority levels instead of the number of elements, requiring more FIFOs and a larger priority encoder.

Brown [24] proposed the calendar queue, similar to the bucket sorting algorithm, operating on an array of lists that contains future events. It is designed to operate with O(1) average performance, but poorly performs with changing priority distribution. Also, extensive hardware support is required for larger priority values.

Sivaraman *et al.* [72] proposed the PIFO queue. A PIFO is a PQ that allows elements to be enqueued into an arbitrary position according to the elements ranks (the scheduling order or time), while dequeued elements are always from the head. The sorting algorithm, called flow scheduler, enables O(1) performance. However, extensive hardware support is required due to the full ordering of the queue elements, compared to the partial sort in Moon's work and

in our design. PIFO manages to enqueue, dequeue, and replace in the correct order and in constant time.

Ioannou and Katevenis [41] proposed a pipelined heap manager architecture that exploits a classical heap data structure (binary tree), while Bhagwan and Lin [15] proposed a pipelined heap (p-heap) architecture (which is similar to a binary heap). These two implementations of pipelined PQs offer scalability and achieve high throughput, but at the cost of increased hardware complexity and performance degradation for larger priority values and queue depths.

Table 3.2 summarizes the expected theoretical results of some PQs already reported in the literature, with their expected and worst case behavior for enqueue and dequeue operations. More details are given in [57] and [68].

Table 3.2 Expected theoretical performance for different PQs

Queue Type	**Enqueue (expected, worst case)**	**Dequeue (expected, worst case)**
Calender queue	O(1), O(n)	O(1), O(n)
Skew heap	O(log(n)), O(n)	O(log(n)), O(n)
Implicit binary heap	O(1), O(log(n))	O(log(n))
Skip lists	O(log(n)), O(n)	O(1)
Splay tree	O(log(n)), O(n)	O(1)
Binary search tree	O(log(n)), O(n)	O(log(n)), O(n)
Systolic array	O(1)	O(1)
Shift register	O(1)	O(1)
Binary heap	O(1), O(log(n))	O(log(n))
PIFO queue	O(1)	O(1)

3.3 Traffic Manager Architecture

In this section, we present a generic TM architecture and its operations. Then, we describe the modules from which it is composed.

3.3.1 Traffic Manager Overview

Traffic management allows bandwidth management, prioritizing, and regulating the outgoing traffic through the enforcement of service-level agreements (SLAs). An SLA defines the requirements that a network must meet for a specified customer or service, and it must be ensured, as a subscriber must get the level of service that was agreed upon with the service provider. The relevant criteria include, but are not limited to the performance metrics, such as guaranteed bandwidth, end-to-end delay, and jitter.

A generic TM in a line card (switch, router, etc.) is depicted in Figure 3.1. In the flowthrough mode, the TM is in the data path. In the look-aside mode, the TM is outside the data path and communicates only with the packet processor, acting as a coprocessor. Generally, TMs reside on the line card next to the switch fabric interface, as they implement the output queuing necessary for the switch fabric or manage the packet buffers of the packet processor.

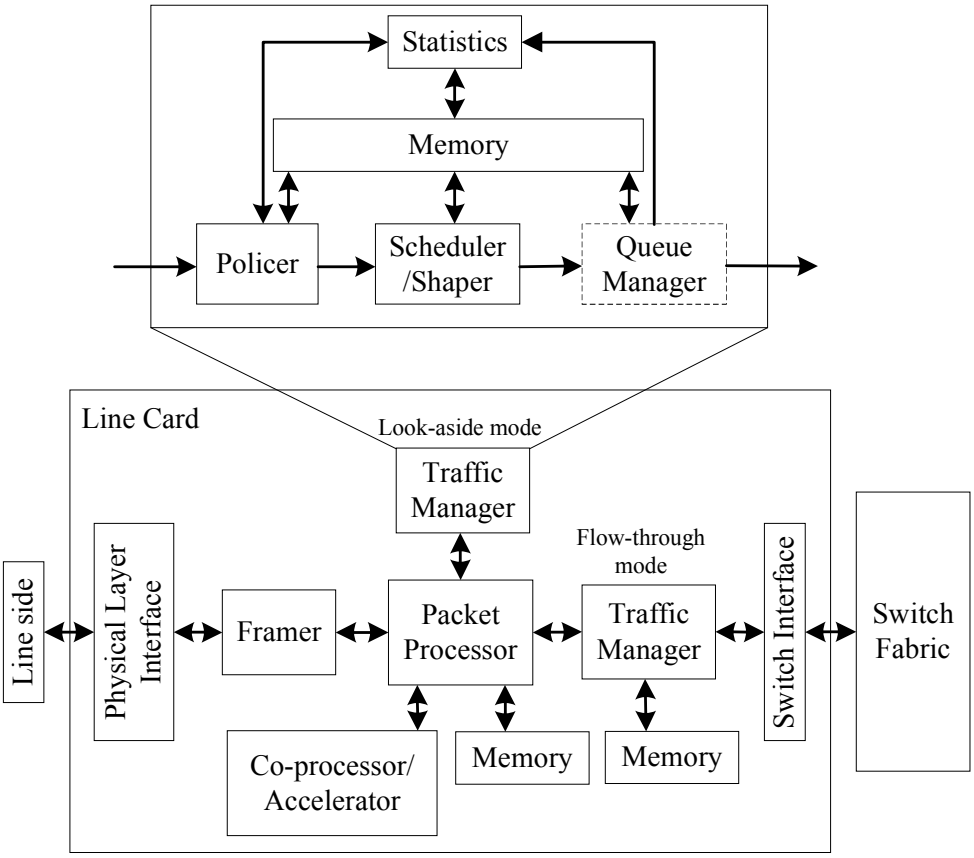


Figure 3.1 Generic architecture around the TM on a line card. This paper focuses on the queue manager block.

3.3.2 Structural Design

A packet processor is used to classify the data traffic to flows (flow tagging) prior its entry into the TM especially in the look-aside mode. A crude definition of a flow is a set of packets associated with a client of the infrastructure provider. These packets may be classified according to their header information. For example, packet classification is done by the packet processor using the five-tuple header information (source and destination IP, source and destination port, and protocol). The classified data traffic allows the TM to prioritize and decide how packets should be scheduled (i.e., when they should be sent to the switch

fabric), how traffic should be shaped when sending packets onto the network, and which appropriate actions to take, for example, drop, retransmit, or forward.

3.3.2.1 Traffic Manager Operations

Traffic scheduling ensures, during times of congestion, that each port and each class of service (CoS) gets its fair share of bandwidth. The scheduler interacts with the QM block, notifying it of scheduling events. Packet congestion can cause severe network problems, including throughput degradation, increased delay, and high packet loss rates. Congestion management can improve network congestion by intelligently dropping packets.

The policer makes decisions on which packets to drop in order to prevent queue overflow and congestion. The shaper enforces packets to follow a specific network pattern by adding delays. The shaper imposes temporary delays to the outgoing traffic to ensure it fits a specific profile (link usage, bandwidth, etc.). During the TM operation, different statistics are being gathered for ingress and egress traffic in terms of received and transmitted packets, the number of discarded packets, etc. These data are stored for future analysis or system diagnosis.

The central piece of the TM is the QM. It maintains the packet priority sorted at link-speed using only the packet descriptor identification (PDI) [86]. The PDI enables packets to be located in the network through small metadata that can be processed in the data plane, while the entire packet is stored outside the TM in the packet buffer. This provides fast queue management with reduced buffering delays. The QM is responsible for packet's enqueue and dequeue or both at the same time. In this paper, we focus on the QM module in the TM of Figure 3.1.

3.3.2.2 Packet Scheduling

The scheduler is responsible to tag, in the packet PDI, the new received packets, with a priority according to the scheduler policy. The PDI contains a priority field with 32 or 16 bits, the packet size in bytes expressed on 16 bits (to support any standard Internet packet size), the packet address location also expressed on 16 bits, and it may also contain other relevant attributes (see Figure 3.2). This priority tagging may represent the expected departure time from the QM, as given by a scheduling algorithm [7, 12]. Also, this priority tagging may represent the CoS (voice, video, signaling, transactional data, network management, basic service, and low priority) for each packet, such that each classified packet corresponding to a flow is given a priority according to its respective traffic class. Packet classification and

scheduling are not discussed further in this work, as we focus on QM functionalities with the proposed SIMD PQ architecture.

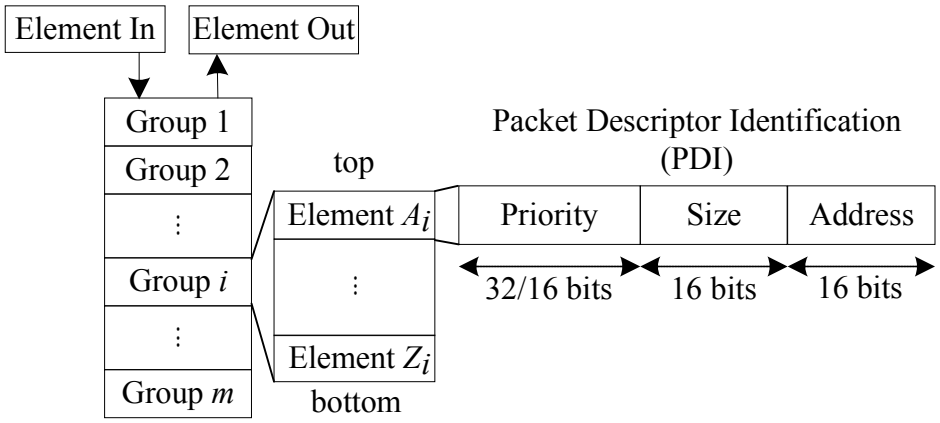


Figure 3.2 SIMD register-array hardware PQ content.

3.3.2.3 Queue Manager

The QM presented in this paper performs enqueue operations of incoming PDIs. Also, the QM produces as outputs the PDIs of the packets that should be sent back to the network through either dequeue or replace operations. More details about the QM architecture are given in Section 3.4. For convenience, the term "packet PDI" is shortened to simply a "packet" in the next sections. It is worth noting that the QM's hardware PQ is composed of groups, each being connected with its adjacent groups, and each independently applying in parallel a common operation on its data. This architecture is register-based SIMD, with only local data interconnects, and a short broadcasted instruction. Thus, this would also qualify the proposed QM as a systolic architecture by some definitions in the literature. In the proposed QM, a fast hardware SIMD PQ is used to sort the packets from highest to lowest in priority, namely, in ascending order.

3.4 Basic Operations of the SIMD Priority Queue

In general, PQs used in network devices have two basic operations: enqueue and dequeue. An enqueue inserts an element into the queue with its priority. A dequeue extracts the top or highest priority element, and removes it from the queue. In this paper, a dequeue–enqueue operation, or simply a replace operation, is considered as a third basic operation. Most works in the literature consider only the two first basic operations [1, 27, 46, 57, 86], but

a few considered the third operation [15, 39, 41] achieving a higher throughput with better scalability.

In this paper, our proposed SIMD hardware PQ supports the following operations.

1. Enqueue (insert): A new element is inserted into the queue that is combined (partly sorted) with existing elements to restore the queue invariants (see Section 3.4.3).
2. Dequeue (extract min): The highest priority element is removed, and remaining elements are partly sorted to restore the queue invariants (see Section 3.4.3).
3. Replace (extract min with insertion): Similar to combined dequeue–enqueue operations, after which the number of elements inside the queue does not change. This operation is simultaneous for the insert and extract-min elements (see Section 3.4.2) while respecting the queue invariants (see Section 3.4.3).

The PQ behaves differently according to the operation to perform (the instruction). It is divided into m groups (see Figure 3.2); a group contains N packets $A_g \dots Z_g$, where g is the group number, A_g and Z_g represent the min and max elements, respectively, and a subset $\mathcal{S}$ containing the remaining elements in any order, namely, a group X_i contains N elements $\{A_i, \mathcal{S}_i, Z_i\}$ with $\mathcal{S}_i = \{X_i \setminus \{\min X_i, \max X_i\}\}$. The letters $A \dots Z$ are used for generality regardless of the actual number of packets except in examples where N is known.

This architecture is based on the work we presented in [12], extended to add a third basic operation and generalizing to N packets in each group. Also, a proof of correct ordering of the queue elements is provided. For convenience, a queue supporting only the same operations as the previously reported architecture [12], without replace operation, is called an original PQ (OPQ).

3.4.1 Enqueue and Dequeue Operations

The algorithm of the OPQ is a combination of insert-and-sort or extract-and-sort for enqueue and dequeue operations, respectively. At every clock cycle, the queue accepts a new entry or returns the packet with the lowest priority value (the highest in priority). Packet movements obey the algorithm depicted in Figure 3.3(a) and (b) for N packets in each group, see the Appendix A for the notation.

In Figure 3.3, Element Out is always connected to A_1 (top element) to reduce dequeue latency, but it is considered valid only on a dequeue or replace, not on an enqueue or no operation. The PQ just executes the same operation for all groups in parallel. All groups are ordered

Element Out = A_1 (always, but not relevant on enqueue)

For all groups i (i = 1, 2, … m) :

a) On enqueue operation:
group $i \leftarrow order\{ Z_{i-1}, A_i, \mathcal{S}_i \}$

b) On dequeue operation:
group $i \leftarrow order\{ \mathcal{S}_i, Z_i, A_{i+1} \}$

c) On replace operation:
group $i \leftarrow order\{\max\{Z_{i-1}, A_i\}, \mathcal{S}_i, \min\{Z_i, A_{i+1}\}\}$

Where $order\ X = \langle \min X, \mathcal{S}, \max X \rangle$,
$\mathcal{S} = X \setminus \{ \min X, \max X \}$,
Z_0 is the incoming packet (Element In),
A_{m+1} is the "none" packet equivalent to empty cell in the queue which must compare as greater (>) to any valid packet,
$\max\{Z_0, A_1\}$ is defined as Z_0 in c) since A_1 is dequeued.

Figure 3.3 Hardware PQ algorithm.

at the same time such that the OPQ enables independent enqueue and dequeue in constant time, regardless of the number of groups. Our implementation (see Section 3.6) does it in a single clock cycle for different queue depths and group sizes.

Note that the algorithm is well defined for any group size $N \geq 2$ and order is equivalent to fully sorting the elements of X only when $N \leq 3$. The unordered set has a single possible order in that case, it has 0 or 1 element for $N = 2$ and 3, respectively.

3.4.2 Replace Operation

An augmented PQ (APQ) is proposed to support the OPQ functionalities with the addition of a combined dequeue–enqueue operation, or simply replace, in the same clock cycle. In the case where both enqueue and dequeue must be performed on the OPQ with only two basic operations, the enqueue operation is prioritized over the dequeue, as to not lose the incoming packet. It should be noted that in case the OPQ is full, the lowest priority element is dropped as a consequence of the enqueuing. To overcome this issue of delaying the dequeue until no enqueue operation is activated in the same cycle, the third basic operation (replace) is proposed to deal with this case.

The algorithm of this APQ is a combination of insert, extract, and order for replace operation, in addition to the support of the enqueue and dequeue operations. The queue accepts a new

entry and returns the packet with the highest priority at the same time. It does so correctly on a data set composed of a combination of the new entry combined with the current queue content. This can be done at every cycle (see Section 3.6). Packet movements obey the algorithm specified in Figure 3.3 (c). Note that for the last group m, the definition of A_{m+1} and the comparison operator implies that min $\{Z_i, A_{i+1}\} = Z_i$. For enqueue operation, this last element is dropped when the PQ is full, but in a replace operation, no element is dropped as one element is dequeued and another is enqueued simultaneously.

Note that $\min\{A, B\}$ and $\max\{A, B\}$ functions must be defined such that one will return A and the other will return B, even when A and B are considered equal by the priority comparison operator.

An illustrative example of packet priorities movement in the OPQ and APQ is shown in Figure 3.4 for few cycles, assuming the exemplary case of three packets in each group. Initially, the PQ contains empty cells. Empty cells priorities are represented by the maximum value of the priority. While time elapses cycle after cycle, the content of the PQ groups is displayed. It is of interest that the highest priority elements (smallest) remain close to the first groups, ready to exit, whereas the elements with lower priorities (largest) tend to migrate to the right of the queue. Meanwhile, it is worth noting that for the same example (see Figure 3.4), the OPQ that does not support replace operation took more cycles and more storage elements as compared to the APQ.

3.4.3 Proof of Validity of the OPQ and APQ Behaviors

Queue invariants are provided to prove the correct functionality of the hardware PQ during enqueue, dequeue, and replace operations. The first two invariants will prove that the top element of the queue is always the highest priority one with no invariant ordering violation. The third invariant is provided to ensure that a drop may occur only in the situation where all the queue groups are full.

For all groups i with $i = 1, 2 \ldots m$ in the PQ, where each group contains N packets $\{A_i, \mathcal{S}_i, Z_i\}$, where $\mathcal{S}_i$ is a subset containing the remaining $N - 2$ elements of group i in any order.

1. Invariant 1: A_i and Z_i are, respectively, the highest and lowest priority elements in the group i. Note that in our case, the highest priority is the smallest value. That is

$$A_i = \min\{A_i, \mathcal{S}_i, Z_i\} \text{ and } Z_i = \max\{A_i, \mathcal{S}_i, Z_i\} \tag{3.1}$$

   where $\{A_i, \mathcal{S}_i, Z_i\} = \{A_i, Z_i\} \cup \mathcal{S}_i$.

	Cycle	In	Out	Group 1			Group 2			Group 3		
Insert	1	5		5								
Insert	2	7		5	7							
Insert	3	6		5	6	7						
Insert	4	2		2	5	6	7					
Extract	5		2	5	6	7						
Insert	6	1		1	5	6	7					
Insert	7	4		1	4	5	6	7				
Insert	8	0		0	1	4	5	6	7			
Insert	9	9		0	1	9	4	5	6	7		
Extract	10		0	1	4	9	5	6	7			
Extract	11		1	4	5	9	6	7				
Extract	12		4	5	6	9	7					
Extract	13		5	6	7	9						
Extract	14		6	7	9							

(a)

	Cycle	In	Out	Group 1			Group 2			Group 3		
Insert	1	5		5								
Insert	2	7		5	7							
Insert	3	6		5	6	7						
Insert	4	2		2	5	6	7					
Extract	5		2	5	6	7						
Replace	6	1	5	1	6	7						
Replace	7	4	1	4	6	7						
Replace	8	0	4	0	6	7						
Replace	9	9	0	6	7	9						
Extract	10		6	7	9							

(b)

Figure 3.4 Example of packet priorities movement in (a) OPQ—packet priorities movement in the original SIMD and (b) APQ—packet priorities movement in the augmented SIMD PQ for few cycles.

2. Invariant 2: Except Z_i, all elements in group i are of higher or equal priority than the first element in group $i+1$. That is

$$\max\{A_i, \mathcal{S}_i\} \leq A_{i+1}. \tag{3.2}$$

   Invariant 2 implies that $A_1 \leq A_2 \leq \ldots \leq A_m$, and by invariant 1, these A_i's are the minimum in their respective groups. Thus, A_1 is the smallest of all values. So, the top element of the first group is the highest priority one in the PQ.

3. Invariant 3: A group i contains valid elements only if all the preceding groups are full.

We need to prove these invariants are preserved by the algorithm specified in Figure 3.3. In the proof, we define that in the following.

a) $A_i \ldots Z_i$ are the elements in group i before the operation, A_i being the first element of the vector, Z_i being the last.

b) G_i is the set of elements passed to the *order* function for group i in the algorithm (see Figure 3.3).

c) $A'_i \ldots Z'_i$ are the elements after the operation, thus the result of the *order* function.

We will prove that if $A_i \ldots Z_i$ satisfy the invariants, $A'_i \ldots Z'_i$ will also satisfy them. The initial state is an empty queue, where all elements in the queue compare equal to each other (represented by the maximum value of the priority), satisfying the invariants. We will prove by induction that the algorithm preserves those invariants for all operations.

3.4.3.1 Proof for Invariant 1

All three operations (enqueue, dequeue, and replace) do

$$\begin{aligned} \left\langle A'_i, \mathcal{S}'_i, Z'_i \right\rangle &= order\ G_i \\ &= \langle \min G_i, G_i \setminus \{\min G_i, \max G_i\}, \max G_i \rangle. \end{aligned} \tag{3.3}$$

Thus, $A'_i = \min G_i$, $Z'_i = \max G_i$ and the other elements are the remaining elements of G_i in any order represented with the subset $\mathcal{S}'_i$. After the operation, invariant 1 is thus satisfied regardless of whether invariants were satisfied or not before the operation.

3.4.3.2 Proof for Invariant 2

Supposing invariants are satisfied on $A_i \ldots Z_i$, from (3.2) we have to verify whether $\max\{A'_i, \mathcal{S}'_i\} \leq A'_{i+1}$ for the three operations. By (3.3), A'_i and $\mathcal{S}'_i$ represent all elements of G_i except its max, thus $\max\{A'_i, \mathcal{S}'_i\}$ is the second largest element in G_i. We thus define 2nd max X as the second largest element in set X. Also by (3.3), $A'_{i+1} = \min G_{i+1}$, the above verification is equivalent to

$$\text{2nd } \max G_i \leq \min G_{i+1} \tag{3.4}$$

In (3.4), as G_i is defined in terms of $A_i \ldots Z_i$, not $A'_i \ldots Z'_i$, by the induction hypothesis, we can use (3.1) and (3.2) on these variables. The following property will also be used:

$$\text{2nd max } X \leq \max(X \setminus \text{any single element}) \tag{3.5}$$

In (3.5), if the single element removed was not the max of X, the max remains the same, and 2nd max $X \leq$ max X by definition, and if it was the max of X, 2nd max $X = \max(X \setminus \max X)$ also by definition, proving (3.5).

a) Proof for the enqueue operation

$G_i = \{Z_{i-1}, A_i, \mathcal{S}_i\}$	by Figure 3.3(a)	.a
$\text{2nd}\max G_i \leq \max\{A_i, \mathcal{S}_i\}$	by .a into (3.5)	.b
$\leq Z_i$	by (3.1) on .b	.c
$\leq A_{i+1}$	by (3.2) on .b	.d
$\leq \min\{A_{i+1}, \mathcal{S}_{i+1}\}$	by (3.1) on .d	.e
$\leq \min\{Z_i, A_{i+1}, \mathcal{S}_{i+1}\}$	by merging .c, .e	.f
$\leq \min G_{i+1}$	by .a on .f	.g

By .g, we verified (3.4), thus invariant 2 is preserved.

b) Proof for the dequeue operation

$G_i = \{\mathcal{S}_i, Z_i, A_{i+1}\}$	by Figure 3.3(b)	.a
$\text{2nd max } G_i \leq \max\{\mathcal{S}_i, A_{i+1}\}$	by .a into (3.5)	.b

$$
\begin{aligned}
&\text{''} && \leq A_{i+1} \quad \text{by (3.2) on .b} && \text{.c}\\
&\text{''} && \leq \min\{\mathcal{S}_{i+1}, Z_{i+1}\} \quad \text{by (3.1) on .c} && \text{.d}\\
&\text{''} && \leq A_{i+2} \quad \text{by (3.2) on .c} && \text{.e}\\
&\text{''} && \leq \min\{\mathcal{S}_{i+1}, Z_{i+1}, A_{i+2}\} \quad \text{by merging .d, .e} && \text{.f}\\
&\text{''} && \leq \min G_{i+1} \quad \text{by .a on .f} && \text{.g}
\end{aligned}
$$

By .g, we verified (3.4), thus invariant 2 is preserved.

c) Proof for the replace operation

$$G_i = \{\max\{Z_{i-1}, A_i\}, \mathcal{S}_i, \min\{Z_i, A_{i+1}\}\} \quad \text{by Figure 3.3(c)} \qquad \text{.a}$$

Left part of (3.4): 2nd max G_i

$$
\begin{aligned}
\text{2nd max } G_i \leq \max\{\mathcal{S}_i, \min\{Z_i, A_{i+1}\}\} \quad & \text{by .a into (3.5)} && \text{.b}\\
\max\{\mathcal{S}_i\} \leq A_{i+1} \quad & \text{by (3.2)} && \text{.c}\\
\min\{Z_i, A_{i+1}\} \leq A_{i+1} \quad & \text{by definition of min} && \text{.d}\\
\max\{\mathcal{S}_i, \min\{Z_i, A_{i+1}\}\} \leq A_{i+1} \quad & \text{by merging .c, .d} && \text{.e}\\
\text{2nd max } G_i \leq A_{i+1} \quad & \text{by .e on .b} && \text{.f}
\end{aligned}
$$

Right part of (3.4): min G_{i+1}

$$
\begin{aligned}
\min G_{i+1} &= \min\{\max\{Z_i,\ A_{i+1}\},\ \mathcal{S}_{i+1}, \min\{Z_{i+1},\ A_{i+2}\}\} \quad \text{by .a} && \text{.g}\\
\text{''} &= \min\{\max\{Z_i, A_{i+1}\},\ \mathcal{S}_{i+1}, Z_{i+1}, A_{i+2}\} \quad \text{by associativity on .g} && \text{.h}\\
A_{i+1} &\leq \max\{Z_i, A_{i+1}\} \quad \text{by definition of max} && \text{.i}\\
\text{''} &\leq \min\{\mathcal{S}_{i+1}, Z_{i+1}\} \quad \text{by (3.1)} && \text{.j}\\
\text{''} &\leq A_{i+2} \quad \text{by (3.2)} && \text{.k}\\
\text{''} &\leq \min G_{i+1} \quad \text{by merging .i-.k into .h} && \text{.m}
\end{aligned}
$$

By .f and .m, we verified (3.4), thus invariant 2 is preserved.

Note that the above proof does not use A_1 for valid values of i $(1 \ldots m)$, thus the special definition of $\max\{Z_0, A_1\}$ at the bottom of Figure 3.3 has no influence on the proof.

We have thus proven that the algorithm preserves the invariants 1 and 2. For the algorithm to be proven correct, we also need to verify that the inserted elements are correctly conserved

in the queue, not deleted nor duplicated. In Figure 3.3, the *order* function clearly keeps all elements without duplication if the min and max removed to make $\mathcal{S}$ are the same as those placed in the first and last elements (remember that elements can have similar priorities, and thus compare as equal in the ordering, but having different associated metadata).

On dequeue [Figure 3.3(b)]: A_1, the outgoing element, is correctly removed; A_{i+1} goes into group i, other elements stay in the same group; group m has an empty cell.

On replace [Figure 3.3(c)]: A_1, the outgoing element, is correctly removed because of the special case at the bottom of Figure 3.3; Z_i and A_{i+1} are in a min (in G_i) and a max (in G_{i+1}), and this will keep both element by the way we defined the min/max pair; for G_m, $\min\{Z_m, A_{m+1}\}$ correctly keeps Z_m if it is valid, by the definition of A_{m+1}.

On enqueue [Figure 3.3(a)]: Z_0, the incoming element, enters in group 1; Z_i goes into group $i+1$, other elements stay in the same group; Z_m is dropped. It is important to show that Z_m will only contain a valid element when the queue is full, and thus, it would be required to drop an element during an enqueue.

3.4.3.3 Proof for Invariant 3

From invariant 1, where A_i and Z_i are, respectively, the smallest and largest in group i, and definition of "none" packet/element (bottom of Figure 3.3) which states they compare as greater than any valid element, we deduce two remarks as follows.

Remark 1: A_i is valid if and only if group i contains at least one valid element.

Remark 2: Z_i is valid if and only if group i is full.

a) Proof for the enqueue operation

On enqueue [Figure 3.3(a)]: $\langle A'_i, \mathcal{S}'_i, Z'_i \rangle = order\{Z_{i-1}, A_i, \mathcal{S}_i\}$. For group i to contain valid elements after the operation, two cases must be considered:

Case 1: Group i was not empty, thus A_i is valid by Remark 1, and A_i being still in group i after the operation, A'_i is also valid by Remark 1. By invariant 3 (induction hypothesis), groups $1 \ldots i-1$ were full ($A_1 \ldots Z_{i-1}$ are valid), and Z_0 is valid by definition of enqueue. Thus, groups preceding i are full ($A'_1 \ldots Z'_{i-1}$ being a reordering of valid $Z_0, A_1 \ldots \mathcal{S}_{i-1}$). Therefore, invariant 3 is preserved.

Case 2: Group i was empty, but Z_{i-1} is valid thus group $i-1$ was full by Remark 2. $A_1 \ldots Z_{i-2}$ are valid by invariant 3 and $A_{i-1} \ldots Z_{i-1}$ are valid as group $i-1$ was full, thus, as in previous case, it implies that invariant 3 is preserved.

b) Proof for the dequeue operation

On dequeue [Figure 3.3(b)], A_{m+1} is the "none" element entering into the last group m per dequeue operation, with $\langle A'_i \ldots Z'_i \rangle = order\{S_i, Z_i, A_{i+1}\}$.

For group i to contain valid elements after the operation, at least one element in the set passed to order must be valid. As A_{i+1} cannot be valid without $A_1 \ldots Z_i$ being valid (induction hypothesis), at least one element of $\{\mathcal{S}_i, Z_i\}$ is valid, and by Remark 1, A_i is valid. By induction hypothesis, groups $1 \ldots i-1$ were full ($A_1 \ldots Z_{i-1}$ are valid). Thus, groups preceding i are full ($A'_1 \ldots Z'_{i-1}$ being a reordering of valid $\mathcal{S}_1 \ldots A_i$), and invariant 3 is preserved.

c) Proof for the replace operation

On replace [Figure 3.3(c)], $\langle A'_i \ldots Z'_i \rangle = order\{\max\{Z_{i-1}, A_i\}, \mathcal{S}_i, \min\{Z_i, A_{i+1}\}\}$. Two cases must be considered for group i before the operation as follows.

Case 1: Group i was not empty, by Remark 1, A_i is valid. By invariant 3, groups $1 \ldots i-1$ were full ($A_1 \ldots Z_{i-1}$ are valid), and Z_0 is valid by definition of replace. Thus, groups preceding group i are still full after the operation ($A'_1 \ldots Z'_{i-1}$ being a reordering of valid $Z_0, A_1 \ldots A_i$), and invariant 3 is preserved.

Case 2: Group i was empty. By Remark 1, A_i is empty, group i remains empty after the operation as the max $\{Z_{i-1}, A_i\}$ returns always a "none/empty" element (A_i). Thus, invariant 3 is preserved.

If Z_m is valid, group m is full by Remark 2, and groups $1 \ldots m-1$ are full by invariant 3, meaning that the PQ is full. Therefore, an element (Z_m) will be dropped only in the case of an enqueue operation when the PQ is full.

We note that, from invariants 1 and 2, we have

$$A_1 \leq (\mathcal{S}_1) \leq A_2 \leq (\mathcal{S}_2) \leq A_3 \ldots, \text{ etc.}$$

The only unordered elements are the Z_i's, thus, in the worst case, Z_m is the mth lowest priority element. For a constant queue depth, increasing N reduces m, and the dropped element when enqueuing on a full queue will be of lower priority. Also, if the *order* function fully sorts the elements (which is always true for groups of size $N \leq 3$), the whole queue is sorted except the Z_i's.

3.4.4 Original and Augmented PQs Decision Tree

A decision tree (DT) is used to implement the ordering independently in each group of the PQ. The entire PQ's group elements are evaluated in parallel and at the same time according to the priority of each element. Increasing the number of elements in each group (the space complexity) will impact directly the DT and the overall queue performance (the time comlexity). Also, this impacts the quality of dismissed elements when the queue is full. This tradeoff is detailed in Section 3.4.4.1.

3.4.4.1 Tradeoff (Time Versus Space)

To choose the number of elements in each group, two things should be taken into consideration: performance and quality of dropped elements when the queue is full. The complexity of the proposed DT for sorting the elements, i.e., the *order* function depicted in Figure 3.3, is O(log(N)). This DT belongs to the family of parallel networks for sorting [43, 49]. Figure 3.5 depicts the *order* function scheme for $N = 2$, 4, and 8 elements, respectively.

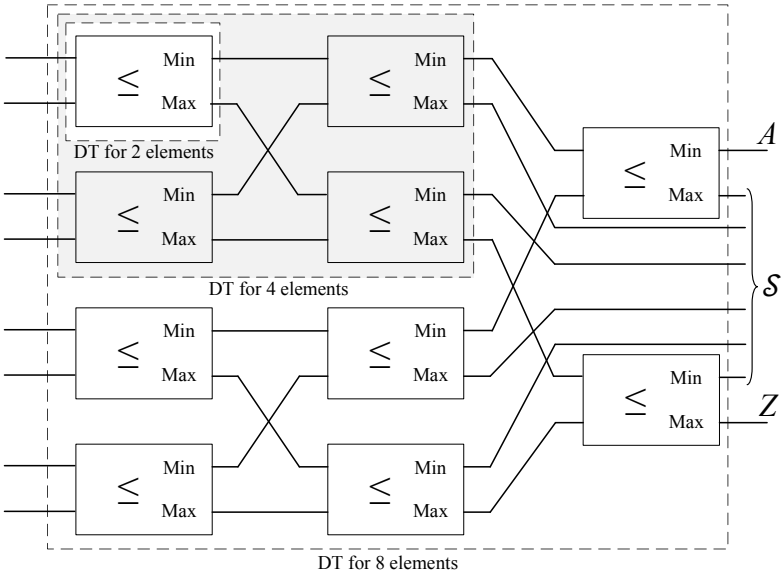


Figure 3.5 Proposed *order* function architecture supporting two, four, and eight elements in each group.

The proposed DT can be further optimized for $N = 3$. In this special case, we take advantage of the present information on already ordered elements in the current group. This DT is depicted in Figure 3.6 for the enqueue, dequeue, and replace operations in each group. On each side of the DT, a specific test is made. For example, the right side is dedicated for the group ordering when only en/dequeue operation is activated, and the left side is for the

replace operation. The order is determined by comparing the priorities of the packets tag present in the different PQ groups using only two comparators. The overall architecture of the proposed hardware SIMD PQ is depicted in Figure 3.7.

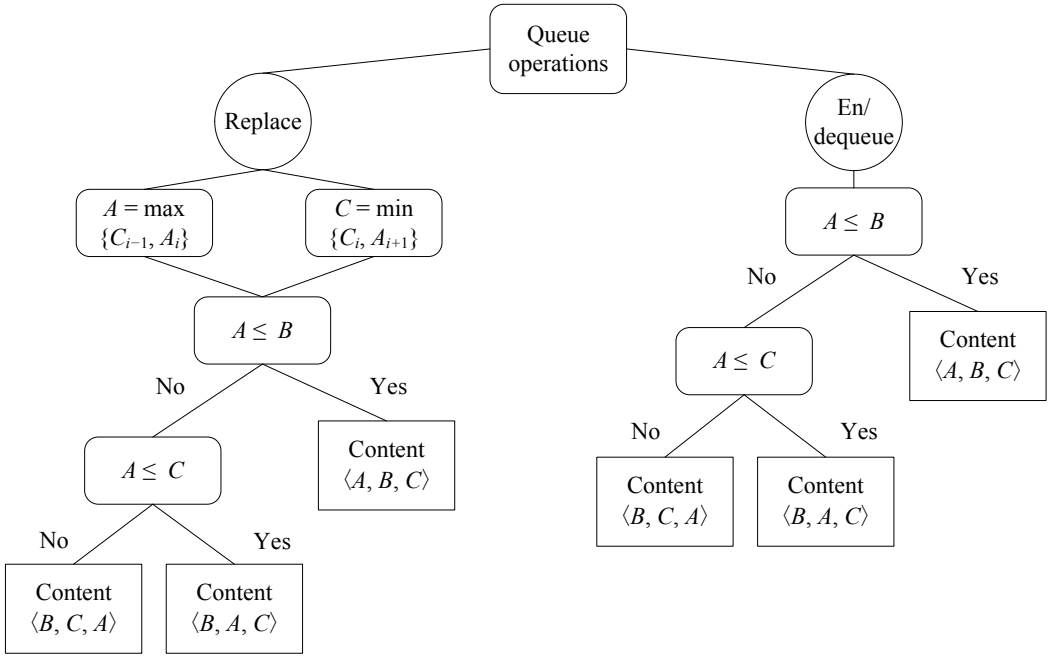


Figure 3.6 Proposed DT giving the result of *order* $\langle A, B, C \rangle$, for three packets in each group of the PQ. Note that we used $\langle\rangle$ instead of $\{\}$ on the *order* function, because it can be optimized relying on known current ordering of elements in the group i. The element coming from another group is called A and the elements from current group are called B and C in priority order in the case of en/dequeue, whereas in replace only A and C are calculated from max/min, respectively, and B is in the current group.

3.4.4.2 Quality of Dismissed Elements

To achieve good performance in terms of latency and number of cycles spent for operations (we target 1 cycle per operation), the proposed architecture is sacrificing two characteristics compared to previous reported approaches depending on the number of groups (m) and the size of the groups (N) for a constant queue depth as follows:

1. Quality of dismissed elements if m is large (the number of elements N in a group is small);

2. Resource usage if m is small (number of elements N in a group is large).

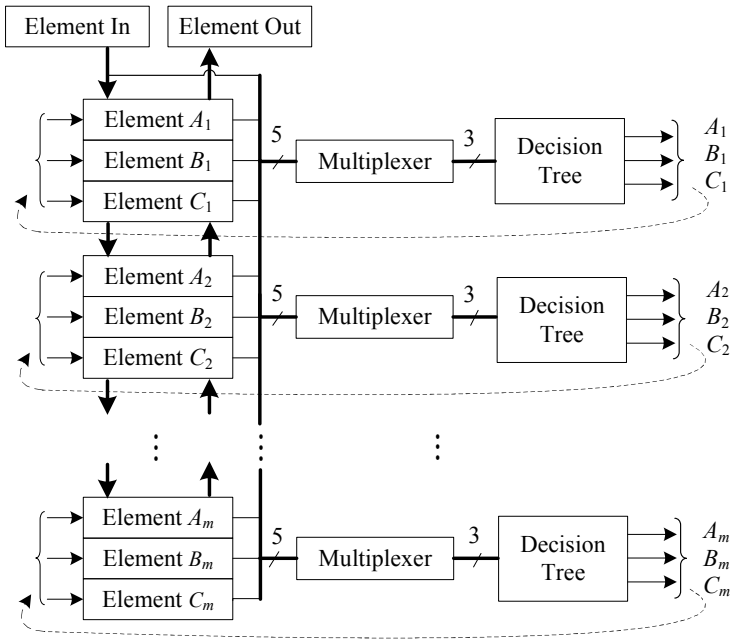


Figure 3.7 SIMD PQ architecture for three packets in each group.

In the worst case, the dismissed element is the mth lower priority element in the queue (the bottom element in the last group m). The queue depth is $m \times N$. So, the quality of the dismissed element is calculated according to the following equation:

$$\text{Quality of dismiss} = \frac{\text{Number of groups}}{\text{Queue depth}} = \frac{m}{m \times N} = \frac{1}{N}. \tag{3.6}$$

For example, $N = 2$, the quality of dismiss is 50%, namely, the dropped element is within the 50% lower priority elements. However, for $N = 64$, this dropped element would be in the 1.56% lower priority elements. So, the higher is N, the best is for the quality, but the performance decreases in O(log(N)). More details about the experimental results are in Section 3.6.

3.5 HLS Design Methodology and Considerations

HLS allows raising the design abstraction level and flexibility of an implementation by automatically generating synthesizable register transfer logic (RTL) from C/C++ models. In addition, exploring the design space using the available directives and constraints allows the user to guide the HLS tool during synthesis. Also, HLS require less design effort, when performing a broad design space exploration as many derivative designs can be obtained with a small incremental effort. Once a suitable specified functionality has been derived, a designer

can focus on the algorithmic design aspects rather than low-level details required when using a hardware description language (HDL).

The first step in any HLS design is the creation of high-level design description of the desired functionality. This description is typically subject to design iterations for refinement (code optimization and enhancement), verification and testing to eliminate bugs, errors, etc. Then, design implementation metrics should be defined such as the target resource usage, desired throughput, clock period, design latency, input-output requirements, etc., which are closely related to the design process, and that are in fact part of the design process. These metrics can be controlled through directives/constraints applied during HLS process. The HLS process can be described in two steps: 1) extraction of data and control paths from the high-level design files and 2) scheduling and binding of the RTL in the hardware, targeting a specific device library. In this paper, we performed all design experimentation with the Vivado HLS tool version 2016.2, while the design was coded in C++.

3.5.1 Design Space Exploration With HLS

Table 3.3 summarizes the results of design space exploration for the OPQ with two elements in each group, while the total specified queue capacity is 64 packets. The metrics used to measure performance in any HLS design are area, latency, and initiation interval (II). Partition directive is used to force the tool to use only logic resources with no BRAMs (on-chip block RAMs) even available in the FPGA. This reduces latency by cutting down the time to memory access. The unroll directive lead to parallelized design producing an output each clock cycle (II = 1), but at higher costs in terms of lookup tables (LUTs) compared to the previous directive. The pipeline directive gives similar results to unroll but it achieves the best clock period. Exploring combinations of the above cited directives with inline for *order* function gives similar results to unroll or pipeline, respectively. However, putting all four directives together in the right place in the code (pipelining the PQ design with II = 1, partition of the queue elements, unrolling the queue groups) gives the best design in terms of resource usage, and performance. These HLS results were generated for all queue configurations (OPQ and APQ) and for different queue depths ranging from 34 up to 1024, while the number of elements in each group varies from $N = 2, 3, \ldots 64$. More details on the experimental results of placement and routing in the FPGA are given in Section 3.6.

3.5.2 Real Traffic Trace Analysis

In order to establish the parameters for the design (especially the queue depth), a detailed analysis was undertaken to find the number of packets that are seen in Internet traffic. This

Table 3.3 HLS design space exploration results of a 64-element OPQ, with $N = 2$ and 64-bit PDI

Optimization	Resources			Performances		
	BRAM	***FFs***	***LUTs***	**Latency**	**II**	**Clock (ns)**
				Min / Max (Cycles)		
Default	6	368	1336	3 – 10	4 – 131	9.19
Partition (1)	0	12267	4992	2 – 95	3 – 96	6.76
Unroll (2)	0	4162	13089	0	1	5.68
Pipeline (3)	0	4098	13025			4.42
(1) + (2) + Inline (4)	0	4162	13089			5.98
(1) + (3)	0	4098	13025			4.42
(2) + (3) + (4)	0	4098	13025			4.42
(1) + (2) + (3) + (4)	0	4098	8621			4.42

was done by examining real traffic traces with different rates, collected by CAIDA from OC-48 and OC-192 representing, respectively, 2.5- and 10-Gb/s links [25]. Table 3.4 depicts the trace characteristics and the rate of packets seen with a monitoring window of 1 ms and 1 s, for 300- and 60-s traces duration for OC-48 and OC-192, respectively. A 1-ms monitoring window is sufficient to satisfy a requirement of high speed as packets are processed in ns time window. From Table 3.4, there are only 75 and 560 packets on average in OC-48 and OC-192 links, respectively, seen in a 1-ms time window. For a queue capacity of 1024 PDIs, it can support today's high-speed links requirement ranging from 2.5 up to 10 Gb/s.

Table 3.4 Traffic trace characteristics and average packets seen in 1-ms and 1-s time intervals for 300- and 60-s duration for OC-48 and OC-192 links CAIDA [25] traces, respectively

Link	Date	Trace Characteristics		Number of Packets	
		Avg. Packet Len. (Byte)	***Avg. Rate (Mbit/s)***	**1 ms**	**1 s**
OC-48	24/04/2003	534	108	25	25248
	15/01/2003	685	324	59	59175
	14/08/2002	571	342	74	74989
OC-192	06/04/2016	833	3744	562	561752
	17/03/2016	601	2073	431	417131
	17/12/2015	662	2463	465	461312

3.6 Experimental Results

In this section, we detail the hardware implementation of our proposed SIMD PQ, as well as its resource usage and achieved performance for different configurations (OPQ and APQ).

Then, comparisons to existing works in the literature are made.

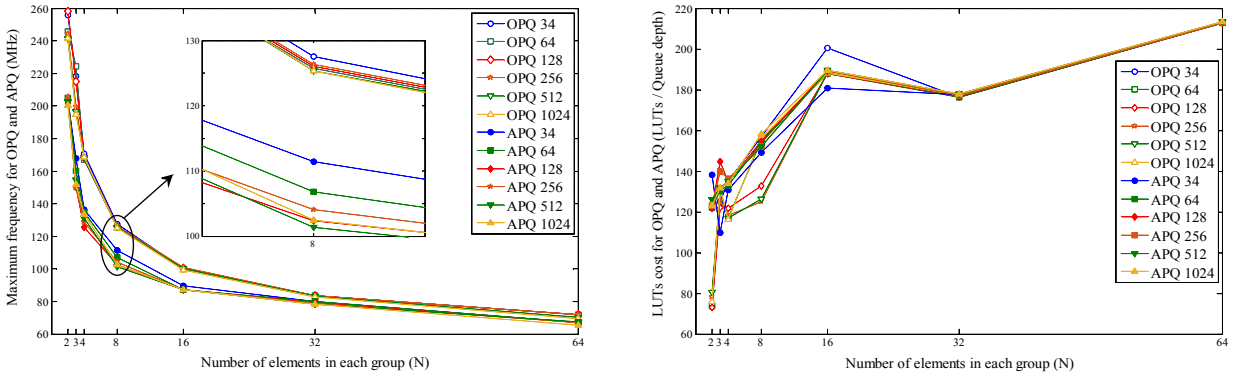
3.6.1 Placement and Routing Results

The proposed hardware SIMD PQ was implemented (placed and routed) on the Xilinx Zynq-7000 ZC706 evaluation board (based on the xc7z045ffg900-2 FPGA). The resource utilization of the implemented hardware PQ architecture for different queue depths are shown in Table 3.5 for OPQ and APQ, with two cases: $N = 2$ and 3 packets in each group. Each hardware PQ element has a 64-bit width, with 32 bit representing the priority, 16 bit for packet size, and 16 bit for the address (see Figure 3.2). The queue depth is varied from 34 to 256, in order to allow comparing with [46]. Also, complexity of 512 and 1024 (1 Ki) deep PQs are summarized in Table 3.5. Other experimental results for different configurations with N varying up to 64 with frequency of operation are shown in Figure 3.8a, and logic resource utilization in Figure 3.8b. Slices utilization for both OPQ and APQ configurations are depicted in Figure 3.9a and 3.9b, respectively.

Table 3.5 Resource utilization of the original and augmented hardware PQs, with 64-bit PDI

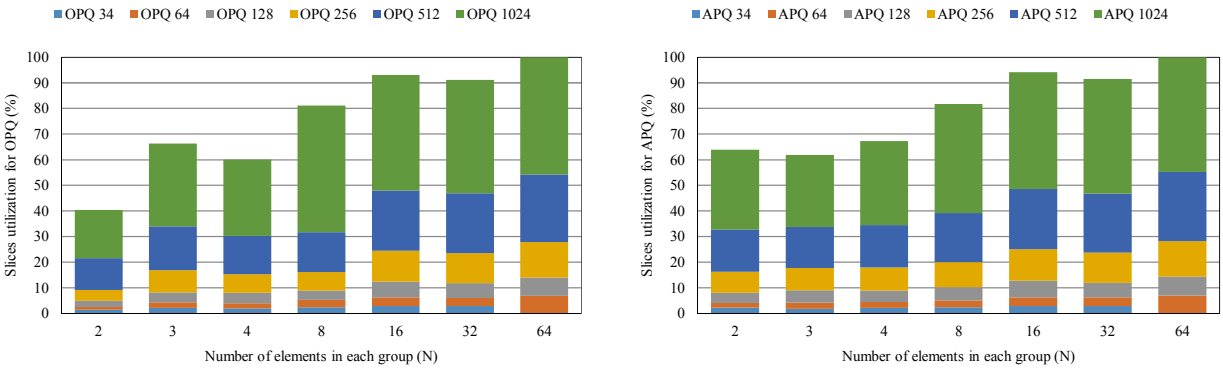
		Queue Depth	34	64	128	256	512	1 Ki
OPQ	$N = 2$	**LUTs**	2494	4719	9362	20003	41257	77896
		FFs	2176	4096	8192	16384	32768	65536
		Utilization	1%	2%	4%	9%	19%	36%
		$\mathbf{F_{max}}$**(MHz)**	256	246	258	244	241	242
	$N = 3$	**LUTs**	4126	8363	15621	32127	66757	134726
		FFs	2112	4032	8064	16320	32640	65472
		Utilization	2%	4%	7%	15%	31%	62%
		$\mathbf{F_{max}}$**(MHz)**	218	224	215	199	196	195
APQ	$N = 2$	**LUTs**	4704	7818	15604	31495	64522	126129
		FFs	2176	4096	8192	16384	32768	65536
		Utilization	2%	4%	7%	14%	30%	58%
		$\mathbf{F_{max}}$**(MHz)**	204	205	202	206	203	201
	$N = 3$	**LUTs**	3630	8349	18523	35806	67514	135068
		FFs	2112	4032	8064	16320	32640	65472
		Utilization	2%	4%	8%	16%	31%	62%
		$\mathbf{F_{max}}$**(MHz)**	168	160	150	150	154	152

When implementing the QM's hardware PQ, only flip-flops (FFs) and LUTs were used to obtain a fast, low latency architecture. The queue is able to take an input and provide an output in the same clock cycle, thus the proposed PQ implementation has a 0-cycle latency. This PQ is capable of performing all the three basic operations in a single clock cycle. Note that as mentioned earlier, the required clock period after placement and routing remains



(a) (b)

Figure 3.8 Experimental results for different queue configurations (OPQ and APQ) with depths ranging from 34 up to 1 Ki. (a) Plots of the maximum frequency of operation for both queues. (b) LUTs cost per queue depth in terms of the number of elements in each group (N). For FFs cost, we obtained a constant 64-bit representing the size of one element (tag) in each group for both OPQ and APQ. These reported results are for 64-bit PDI, with 32-bit priority.



(a) (b)

Figure 3.9 Slices utilization results for different queue configurations. OPQ (left) and APQ (right) histogram with 64-bit PDI.

almost constant regardless of the queue depth, as depicted in Figure 3.8a for different N ranging from 2 up to 64 elements in each group, for both OPQ and APQ, with queue depths varying from 34 up to 1 Ki. The operating frequency reported, after placement and routing (using Vivado 2016.2 with the Explore directive enabled), by the timing analysis tool degrades in O(log(N)) between a maximum 258 and a minimum 69 MHz for the OPQ with a group size of 2 up to 64. For the APQ, similar behavior is observed while the maximum frequency is 206 degrading to 65 MHz.

From Table 3.5 and Figure 3.8b, it is clearly seen that increasing the group size N, leads to an increase in LUTs consumption and not FFs (cost of FFs per elements remained constant and equal to 64-bit, i.e., reflecting the element size). This is due to the required logic in the DT for the additional packets in each group, leading to larger multiplexers and more levels of comparators. Also, the added replace operation to the hardware PQ does increase the LUTs consumption only for small $N < 16$. This LUT usage increase is related to the fact that the replace operation did not require architectural modification on the OPQ. Indeed, it only added a min and max calculation prior to the DT or the *order* function in each group. However, when N increases, the impact of the min/max decreases and the OPQ and APQ LUTs usage converges to nearly the same value when $N \geq 16$.

Figure 3.9 summarizes the slice usage for the OPQ and APQ with different configurations (queue depth and number of elements N). It is of interest that both OPQ and APQ have similar slice usage for $N \geq 16$ (similar to the previous explanation of min/max influence needed in the replace operation). On the other hand, as the APQ is more complex than the OPQ, this min/max influence is mostly seen for $N < 16$. For $N = 3$, the slices usage for different queue capacities in OPQ can be lower compared to the APQ (as for 128 and 256), and it is always larger in the remaining queue capacities, this particular case is only observed for $N = 3$. A lower complexity APQ was only observed for this particular case.

The achieved frequency for the different hardware PQs are also reported as functions of queue depth in Table 3.5 and Figure 3.8a. The achieved throughput in the case where there are two packets in each group with replace (APQ) is 206 Millions packets per second (Mp/s) and in the case of three packets per group, it is 150 Mp/s for a 256 queue capacity. In the case of the OPQ, the throughput is 122 and 99.5 Mp/s for the cases of two and three packets in each group of the PQs, respectively, for similar queue capacity. Noting that the minimum number of operations required to pass a packet through the OPQ is two (enqueue and dequeue), in contrast to APQ which is one (replace). This achieved throughput is stable for the different queue capacities ranging from 34 to 1 Ki and for both queue types. Moreover, even in the worst case performance with $N = 64$, both OPQ and APQ can reach 40 Gb/s throughput for 84-B minimum size Ethernet packets, including minimum size packet of 64 B, preamble and interpacket gap of 20 B.

3.6.2 Comparison With Related Works

When compared to the systolic array in Moon's work [57] (see Table 3.6) supporting only enqueue and dequeue operations, our resource usage in terms of LUTs and FFs is lower. The resource usage results obtained with the proposed architecture (OPQ with 2 packets in each

group) are comparable with the shift-register architecture in terms of FFs. They are lower in terms of LUTs (up to $N = 64$, our architecture remains comparable in terms of FFs), and they are higher than those reported for the hybrid p-heap architecture [46]. The reported design is entirely coded at high-level C++ language as compared to existing architectures coded at low-level in Verilog, VHDL, etc.

Table 3.6 Resource utilization comparison with Kumar *et al.* [46], with 64-bit PDI

Queue Depth	**Shift Register[a]**		**Systolic Array[a]**		**Hybrid P-Heap**		**OPQ Design ($N = 2$)**	
	FFs	***LUTs***	***FFs***	***LUTs***	***FFs***	***LUTs***	***FFs***	***LUTs***
31	2077	4995	3999	8560	906	1411	2176	2494
63	4221	10275	8127	17520	1048	1996	4096	4719
127	8509	20835	NA	NA	1182	2561	8192	9362
255	NA	NA	NA	NA	1330	3161	16384	20003

[a.] Shift register and systolic array architectures implementation are based on the work of Moon [57]

Table 3.7 compares results obtained and reported in the literature with some relevant queue management architectures. The reported throughput of the QMRD [33] system depends on the protocol data unit payload size, while the reported OD-QM [86] results are for 512 active queues, and 64 bytes per packet. Our design is implemented with a total of 1-Ki queue PDIs capacity. The reported throughput is for the worst case egress port speed with 64-B sized packets, while offering 10× throughput improvements (APQ with $N = 2$). It should be noted that our design supports pipelined enqueue, dequeue and replace operations.

Table 3.7 Memory, speed, and throughput comparison with queue management systems, with 64-bit PDI

Queue Management System	**Memory Hierarchy**	**BRAM Complexity**	**Performance**	
			Speed (*MHz*)	**Throughput (*Gb/s*)**
OD-QM [86]	On-chip	56×36kbits	133	8
QMRD [33]	On-chip	389×18kbits	NA	< 9
NPMADE [59]	External SRAM	17×18kbits	125	6.2
APQ Design ($N = 2$)	NA	NA	201	103

The number of cycles between successive dequeue–enqueue (hold) or replace operations, as depicted in Table 3.8, for the OPQ is two clock cycles and only one clock cycle for the APQ supporting the replace. This is less than the FIFO [86] for 256-deep queues, binary heap [46], and p-heap architecture [41]. The reported shift register and systolic architectures in Moon's

work [57] have both a latency of two clock cycles for en/dequeue. The systolic PQ described by Moon et al. [57] is not fully sorted until several cycles due to the fact that only one systolic cell is activated each time, i.e., the lower priority entry is passed to the neighboring block on the next clock cycle. In the case of the shift register proposed by Chandra and Sinnen [27], the performance degrades logarithmically. Compared to the p-heap architecture [41], even though it accept a pipelined operation each clock cycle (except in case of successive deletions), the latency is O(log(n)) in terms of the queue capacity against O(1) time latency for our proposed architecture.

For the PIFO queue [72], we implemented (placed and routed) the sorting data structure used in the PIFO block, called flow scheduler, on the ZC706 FPGA board with 16-bit priority and 32-bit metadata, using the Verilog code provided by the authors. This design supports a total capacity of 1024 elements. It is worth mentioning that this code was intended for a 16-nm standard cell ASIC implementation. Meanwhile, this architecture supports a dequeue–enqueue or replace each cycle. This architecture fully sorts all elements in parallel in a single pass through parallel comparators and encoder to determine the right position (the first 0–1 inversion) in which an incoming packet should be inserted. Both enqueue and dequeue operations require two clock cycles to complete. The FFs cost for PIFO is comparable to our architecture with a total of 58.5k FFs against 49.1k FFs, respectively. However, in terms of LUTs cost, our architecture (APQ with $N \leq 64$ and 32-bit metadata) is similar to the PIFO with 210 LUTs per element [see Figure 3.8b] with 32-bit priority, while the cost in LUTs is only 149 per element for APQ with 16-bit priority. The total LUTs usage for the PIFO architecture is 215k LUTs. This expensive cost for the PIFO is mainly due to the extra logic necessary to fully sort the elements in the PIFO block, while our architecture partially sort the elements in each group, and it is capable to restore the queue invariants (see Section 3.4.3) in a single clock cycle.

Both architectures (OPQ and APQ) are capable of satisfying the invariants property for the entire queue in only one clock cycle (in each cycle all groups are being sorted in parallel). Also, this fixed number of cycles in our design is independent of queue depth unlike the O(log(n)) time for the dequeue operation with the heap [46, 53, 77, 87] and binary heap [39], where n is the number of nodes (keys). The achieved throughput is 151 Mp/s for the OPQ and 250 Mp/s for the APQ with 16-bit priority, and 32-bit metadata, against 76.3 Mp/s as the highest reported throughput in Table 3.8 for Huang's work [39], while having the same FPGA board, Ioannou and Katevenis [41] with 90 Mp/s, and Chandra and Sinnen [27] with 102 Mp/s. The APQ is at least 2.45× faster than the latter architectures. For 32-bit priority with 32-bit metadata, our design achieves 121 and 201 Mp/s for OPQ and APQ, respectively. Moreover, the APQ is 2.0× faster than the reported works. Compared to [15, 39, 41, 57, 86],

Table 3.8 Performance comparison for different PQ architectures

Architecture	**Queue Depth**	**# cycles per Hold Operation**	**clock (ns)**	**Priority ; Metadata / Platform**	**Throughput (Mpps)**
FIFO PA-QM[86]	32	9 (no replace)	2.13 (64 queues)	NA ; 64 / FPGA (Vitex-5)	52.2
			2.16 (128 queues)		51.3
			2.67 (256 queues)		41.7
Shift Register [57]	64	2 (no replace)	13.5	16 ; NA / ASIC	37.0
	256		18.2		27.5
	1 Ki		20.0		25.0
Systolic Array [57]	64	2 (no replace)	20.8	16 ; NA / ASIC	24.0
	256		21.7		23.0
	1 Ki		22.2		22.5
Shift register [27]	256	2 (no replace)	3.76	16 ; 32 / FPGA (Cyclone II)	133
	512		4.17		120
	1 Ki		4.92		102
	2 Ki		6.64		75.3
Hybrid register/binary heap [39]	1 Ki	4 : replace 11 : others	~7.5	13 ; 51 / FPGA (Zynq-7)	Best: 33.3 Worst: 12.1
	2 Ki	2 : replace 11 : others	~8.0		Best: 62.5 Worst: 11.4
	4 Ki	1 : replace 11 : others	13.1		Best: 76.3 Worst: 6.9
	8 Ki	1 : replace 11 : others	15.0		Best: 66.7 Worst: 6.1
Pipelined Heap [41]	16 Ki	2 - 17 (no replace)	5.56	18 ; 14 / ASIC	Best: 90 Worst: 10.6
PIFO [72]	1 Ki	2 : replace 4 : others	13.9	16 ; 32 / FPGA (Zynq-7)	36.0 18.0
Proposed OPQ $N = 2$	1 Ki	2 (no replace)	3.31	16 ; 32 / FPGA (Zynq-7)	151
			4.14	32 ; 32 / FPGA (Zynq-7)	121
Proposed APQ $N = 2$		1: replace 2: others	4.0	16 ; 32 / FPGA (Zynq-7)	Best: 250 Worst: 125
		1: replace 2: others	4.98	32 ; 32 / FPGA (Zynq-7)	Best: 201 Worst: 100

the reported throughput is independent of the queue depth.

Compared to existing NPU solutions like Broadcom [21], Mellanox (EZchip) NPS-400 [32], that can support up to 200 and 400 Gb/s, respectively, with built-in queue management systems, our proposed architecture is scalable in terms of performance for different queue capacities. Using a single FPGA (Zynq-7000), we can support links of 100 Gb/s with 64-B sized packets (32-bit priority with APQ). To scale up to 400 Gb/s and beyond, we can use a larger FPGA, for example, an UltraScale that has more logic resources could accommodate all design requirements, or using many FPGAs in parallel like in a multicard "pizza box" system, and/or some combination of these latters. Moreover, it should be noted that the FPGA solution is more flexible than the one of a fixed and rigid logic of an ASIC chip solution.

3.7 Conclusion

This paper proposed and evaluated a priority queue in the context of flow-based networking in a TM. The proposed QM was entirely coded in C++, and synthesized using Vivado HLS. The resource usage of this implementation is similar to other priority queues in the literature, even though they were coded with low-level languages (Verilog, VHDL, etc.). Meanwhile, the achieved performance is at least 2× better than a comparable priority queue design, with a throughput of 10× faster than reported queue management system work in the literature for 1024 deep queues with 32-bit priority. Also, the achieved latency is in O(1) time for enqueue, dequeue, and replace operations, independent of the queue depth. HLS provides flexibility, rapid prototyping, and faster design space exploration in contrast to low-level hand-written HDL designs.

Future work will focus on integrating the proposed priority queue in a flow-based TM, and on assessing its capabilities and performance in practical high-speed networking systems.

CHAPTER 4 ARTICLE 2: HPQS: A FAST, HIGH-CAPACITY, HYBRID PRIORITY QUEUING SYSTEM FOR HIGH-SPEED NETWORKING DEVICES

This chapter contains the paper entitled "HPQS: A Fast, High-Capacity, Hybrid Priority Queuing System for High-Speed Networking Devices" submitted to the IEEE Access[1] journal. As a second idea, we propose a hybrid priority queuing system for strict priority scheduling and high capacity priority queuing, intended for today's network data planes and high-speed networking devices. The implementation results show that we can support links up to 40 Gb/s with guaranteed performance of one clock cycle per queue operation, while the total implemented capacity can reach up to ½ million packet tags when targeting a ZC706 FPGA board and a XCVU440 Virtex UltraScale device.

Note that for those who read chapter 2, Section 2.2, can skip Section 4.3 of this chapter.

Abstract—In this paper, we present a fast hybrid priority queue architecture intended for scheduling and prioritizing packets in a network data plane. Due to increasing traffic and tight requirements of high-speed networking devices, a high capacity priority queue, with constant latency and guaranteed performance is needed. We aim at reducing latency to best support the upcoming 5G wireless standards. The proposed hybrid priority queuing system (HPQS) enables pipelined queue operations with O(1) time complexity. The proposed architecture is implemented in C++, and is synthesized with the Vivado High-Level Synthesis (HLS) tool. Two configurations are proposed. The first one is intended for scheduling with a multi-queuing system for which implementation results of 64 up to 512 independent queues are reported. The second configuration is intended for large capacity priority queues, that are placed and routed on a ZC706 board and a XCVU440-FLGB2377-3-E Xilinx FPGA supporting a total capacity of ½ million packet tags. The reported results are compared across a range of priority queue depths and performance metrics with existing approaches. The proposed HPQS supports links operating at 40 Gb/s.

[1]I. Benacer, F.-R. Boyer, and Y. Savaria, "HPQS: A Fast, High-Capacity, Hybrid Priority Queuing System for High-Speed Networking Devices," submitted to the IEEE Access in 2019. This work was supported in part by the Natural Sciences and Engineering Research Council of Canada (NSERC), in part by Prompt Québec, in part by Ericsson Research Canada, in part by Mitacs, and in part by Kaloom.

4.1 Introduction

In modern routers, switches, line cards, etc., we find Network Processing Units (NPUs). They provide dedicated processing stages for traffic management and buffering. Traffic management includes policing, scheduling, shaping and queuing. For high-speed network switches and devices, queuing may represent a bottleneck. One of the feasible solutions to reduce queuing latencies is to tag the packets. This tagging will hold concise packet information for fast processing, while the actual packets are buffered independently by the NPU, thus reducing the queuing latencies between the different network processing stages [86].

Priority queues have been used in many applications such as event driven simulation [24], scheduling [84], real-time sorting [15], etc. A priority queue (PQ) can be represented as an abstract data structure that allows insertion and extraction of items in priority order. Different types of PQs have been proposed. In the literature, solutions span between the following: calendar queues [24], binary trees [57], shift registers[16, 27, 57], systolic arrays [12, 57], register-based arrays [39], and binary heaps [15, 39, 41, 46]. However, PQs can be divided in two classes: PQs with O(1) time complexity operations, independently of the queue size (number of nodes), and those with variable processing times.

One of the significant challenges facing network operators and Internet providers is the rising number of connected devices. This sets a need for scheduling, prioritizing packets of different applications, and routing the related traffic in a minimum time with the upcoming next generation cellular communication infrastructure (5G) [62]. Also, many applications must deal with real-time traffic, such as video streaming, voice over Internet protocol (VoIP), online gaming, etc. These applications require quality of service (QoS) guarantees. QoS are quantitative measures of the service provided by the network, for example, the average throughput, end-to-end delay, and packet loss. To provide such QoS for large numbers of connected devices, users, etc., high capacity priority queues must be used to maintain real-time sorting of queue elements at link speeds with guaranteed performance.

In this work, we propose a hybrid priority queuing system (HPQS) with two distinct configurations. The first configuration is intended for strict priority scheduling with distinct queues. The second configuration is a single high capacity queue intended for priority queuing. We present placement and routing results in a ZC706 board and XCVU440 device, the total capacity can reach ½ million packet tags of 16-bit priority keys in a Field Programmable Gate Array (FPGA). The first configuration contains a maximum of 512 independent queues with varying queue's width from 64 up to 1024 32-bit elements. The second configuration is a single high capacity queue with 64-bit elements. The HPQS is proposed for high-speed

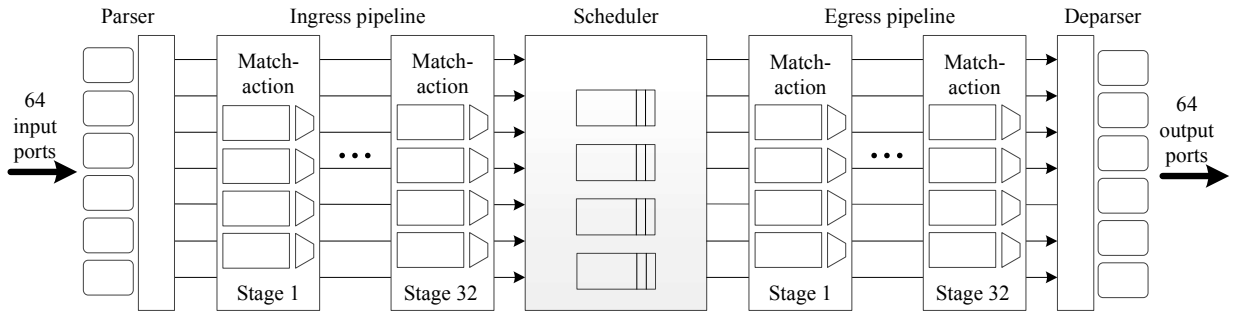


Figure 4.1 A general switch architecture with 64 input/output ports based on [72].

networking devices operating in a constant 1-cycle latency per packet (queue operation) targeting 10 to 40 Gb/s network links. The performance of the proposed HPQS is independent of the PQ capacity. it can be used in different contexts such as traffic managers, task schedulers, sorting, etc. Also, the proposed HPQS architecture is entirely coded in C++, providing easier implementation and more flexibility than some reported works in literature, which use low-level coding, mostly in Verilog, VHDL, and targeting ASIC implementations.

The remainder of this paper is organized as follows. In Section 4.2, we present a general switch architecture. That switch architecture provides a meaningful context where the proposed HQPS would fit. In Section 4.3, we detail some related work on PQs found in the literature. In Section 4.4, the architecture of the proposed HPQS with its different configurations is presented. In Section 4.5, we detail the high-level synthesis (HLS) HPQS design methodology and considerations leading to the best performances. Section 4.6 reports hardware implementation results and comparisons to related work. Finally, Section 4.7 draws conclusions from this work.

4.2 Background

In this section, we present the general architecture of a shared memory switch. Then, we elaborate on the concepts of scheduling and priority queuing.

4.2.1 Network Switches

Today's switches provide various sets of functionalities, from parsing, classification, scheduling and buffering of the network traffic. These functionalities can be supported by transformations applied to the traffic from the moment packets are received on input ports up to their transmission through destination ports. The architecture of a shared memory switch,

with its internal modules such as Broadcom's Trident II series [22], is depicted in Figure 4.1. From the requirements of today's networks, switches must run at line rates of 10 to 40 Gb/s. More details about scheduling and priority queuing are given in the next subsection.

4.2.2 Scheduling and Priority Queuing

Network switches must provide scheduling capabilities (see Figure 4.1). Some of the well-known scheduling algorithms are deficit round robin (DRR), fair queuing [70], and strict priority [36], etc. In this work, we are mainly interested in scheduling through strict priority and priority queuing, while providing large buffering capacity implemented using on-chip memories in an FPGA.

The high capacity priority queue is provided with guaranteed performance that can be used for sorting purposes. This sorting may represent the prioritization of the different class of service (CoS): voice, video, signaling, transactional data, network management, basic service, and low priority for each packet or flow. A flow maybe defined for example from the 5-tuple header information (source and destination IP, source and destination port, and protocol). The priority tag is generated prior entry of the packet tag into the HPQS by the classification stage. It should be noted that packet classification is not discussed further in this paper, as we focus on strict priority scheduling and high capacity priority queuing with the proposed HPQS architecture in FPGA.

4.3 Related Work

Several PQs have been proposed in the literature. These works can be classified as software-based and hardware-based solutions. In software-based solutions, we find mainly heaps and binary search trees [39, 76]. However, these implementations cannot handle large priority queues with high throughput and very low latency, due to the inherent O($\log n$) complexity per queue operation, where n is the number of keys.

Reported solutions for hardware PQs are based on calendar queues [24], binary trees [57], shift registers [16, 27, 57], systolic arrays [12, 57], and binary heaps [15, 39, 41, 46]. Moon [57] analyzed four scalable priority queue architectures based on: FIFOs, binary trees, shift registers and systolic arrays. Moon showed that the shift register architecture suffers from heavy bus loading, and that the systolic array overcomes this problem at the cost of doubling the hardware complexity. Meanwhile, the total capacity previously investigated by Moon is 1024 (1 Ki) elements. Also, the hardware approaches that were adopted limit queue size scalability due to limited resources. This motivated the research reported in the present paper to

explore alternatives for building high capacity priority queues that can offer high throughput and low latency with O(1) time complexity (guaranteed performance). The reported solution is a hybrid PQ. Basically, hybrid PQs combine dedicated hardware approaches extended using on-chip or off-chip memories. In this work, we target to use only on-chip memories available in FPGAs (block RAMs).

Bhagwan [15] and Ioannou [41] proposed hybrid priority queue architectures based on a pipelined heap, i.e., a p-heap (which is similar to a binary heap). However, the proposed priority queue supports en/dequeue operations in O(log n) time against a fixed time for the systolic array and shift register, where n is the number of keys. Also, these two implementations of pipelined PQs offer scalability and achieve high throughput, but at the cost of increased hardware complexity and performance degradation for larger priority values and queue sizes. The reported solutions implemented on ASICs had 64 Ki [41] and 128 Ki [15] as maximum queue capacities. Kumar [46] proposed a hybrid priority queue architecture based on a p-heap implemented on FPGA supporting 8 Ki elements. This architecture can handle size overflow from the hardware queue to the off-chip memory. Huang [39] proposed an improvement to the binary heap architecture. Huang's hybrid PQ combines the best of register-based array and BRAM-tree architectures. It offers a performance close to 1 cycle per replace (simultaneous dequeue-enqueue) operation. In this solution, the total implemented queue capacity is 8 Ki elements when targeting the ZC706 FPGA board.

Zhuang [87] proposed a hybrid PQ system exploiting an SRAM-DRAM-FIFO queue using an input heap, a creation heap and an output heap. The packet priorities are kept in sorted FIFOs called SFIFO queues that are sorted in decreasing order from head to tail. The 3 heaps are built with SRAM, while the SFIFO queues extend the SRAM-based output heap to DRAM. Zhuang validated his proposal using a 0.13 µm technology under CACTI [67] targeting very large capacity and line rates: OC-768 and OC-3072 (40 and 160 Gb/s) while the total expected packet buffering capacity reached 100 million packets.

Chandra [27] proposed an extension of the shift register based PQ of Moon [57] using a software binary heap. For larger queue capacity implementation (up to 2 Ki), the resource consumption increases linearly, while the design frequency reduces logarithmically. This is a limitation for larger queues in terms of achieved performance and required hardware resources. Bloom [16] proposed an exception-based mechanism used to move the data to secondary storage (memory) when the hardware PQ overflows.

Sivaraman [72] proposed the PIFO queue. A PIFO is a PQ that allows elements to be enqueued into an arbitrary position according to the elements ranks (the scheduling order or time), while dequeued elements are always from the head. The sorting algorithm, called flow

scheduler in a PIFO, manages to enqueue, dequeue, and replace in the correct order and in constant time a total capacity of 1 Ki elements.

McLaughlin [53, 54] proposed a packet sorting circuit based on a lookup tree (a trie). This architecture is composed of three main parts: the tree that performs the lookup function with 8 Ki capacity, the translation table which connects the tree to the third part, the tag storage memory. It was implemented as an ASIC using the UMC 130-nm standard cell technology, and the reported PQ had a packet buffering capacity of up to 30 million packets tags.

Wang [77, 78] proposed a succinct priority index in SRAM that can efficiently maintain a real-time sorting of priorities, coupled with a DRAM-based implementation of large packet buffers targeting 40 Gb/s line rate. This complex architecture was not implemented, it was intended for high-performance network processing applications such as advanced per-flow scheduling with QoS guarantee.

Afek [1] proposed a PQ using TCAM/SRAM. This author showed the efficiency of the proposed solution and its advantages over other ASIC designs [53, 54, 87], but its overall rate degrades with larger queue size while targeting 100 Gb/s line rate. Also, Afek presented an estimation of performance with no actual implementation.

Van [75] proposed a high throughput pipelined architecture for tag sorting targeting FPGA with 100 Gb/s line rate. This architecture is based on multi-bit tree and provides constant insert and delete operation requiring 2-clock cycles. The total supported number of packet tags is 8 Ki.

4.4 The Hybrid Priority Queue Architecture

In this section, we present the HPQS architecture. Then, we detail its different queuing models for scheduling and priority queuing, with the supported sorting types.

This work is an extension of a previous related work [13]. The new contributions are as follows:

1. Design of a HPQS that supports two configurations. The first configuration (distinct-queues model) with partial and full sort capability supporting en/dequeue operations. The second configuration (single-queue model) supports a third queue operation (replace). The HPQS throughput can reach 40 Gb/s for minimum sized packets, with guaranteed O(1) time complexity per queue operation (see Section 4.6).

2. Analysis of HPQS operations leading to improvements that allowed matching the performance of hand-written register transfer logic (RTL) codes with an HLS design (see

Section 4.5).

3. Design space exploration under ZC706 FPGA and XCVU440 device for resource (look-up tables and flip flops), performance metrics (throughput, latency, and clock period), and power consumption analysis of the HPQS design (more details are given in Section 4.6.1).

In this work, packets are sorted in ascending order of priority based on PQ presented in [9, 12]. The supported queue operations are enqueue and dequeue in the distinct-queues model (type-1). A third operation, i.e., replace, is also supported in the single-queue model (type-2). An enqueue enables insertion of an element to the HPQS, while a dequeue removes the highest priority element (lowest in priority value). The replace operation allows insertion while extracting the highest priority element. The whole HPQS design is described at high-level using the C++ language. The code is written in a way that allows efficient hardware implementation and prototyping in a FPGA platform.

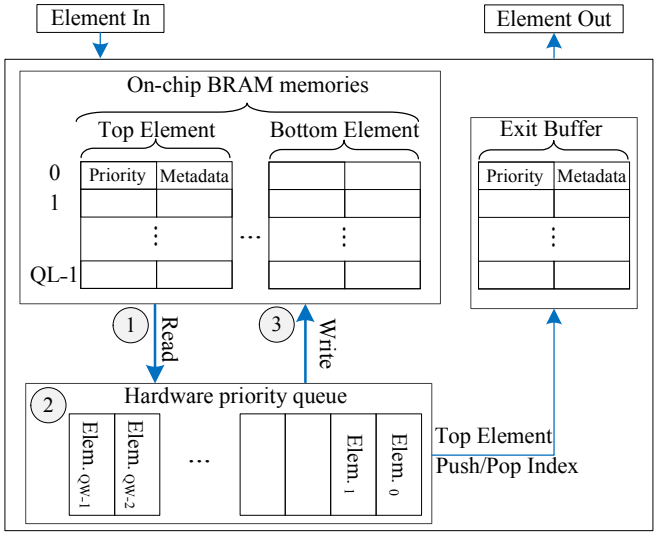


Figure 4.2 The proposed HPQS architecture.

Figure 4.2 depicts the proposed HPQS architecture. The HPQS assumes one input port representing the operation to perform and the pushed packet tag (Element In), and one output port representing the dequeued packet tag (Element Out). The HPQS is devised in three main parts. The first part is the storage area of the queues implemented with on-chip Block RAMs (BRAMs). The second part contains the hardware PQ used to sort out packets in a single clock cycle, its depth represents the HPQS queuing width (QW). The third part is represented by the exit buffer that is holding the top element for each queue line.

4.4.1 Type-1 Distinct-Queues Model

In the type-1 architecture, each queue line (QL) of the storage area represents a distinct queue. The system assumes the highest priority is the top queue, while the lowest is the bottom one. Elements will be dequeued from top to bottom queues according to their occupancy in ascending order of priority values. This represents a strict priority scheduler with distinct queues.

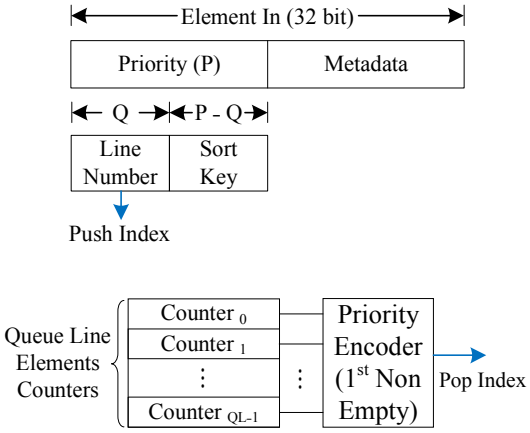


Figure 4.3 Push and pop indices calculation in HPQS type-1 architecture.

When a packet tag is received (Element In) as depicted in Figure 4.3, it contains its priority and required metadata information like the actual packet address, egress destination port, or any other attributes. The configuration of the data type limit can be modified for tag fields. The received packet tag is pushed with reduced priority information into the HPQS (priority information for sorting + push index) that is subsequently sorted in the appropriate queue. In this architecture, the priority key is 16-bit (P), and the word element length is 32-bit. The total number of queues, i.e. QL, is given by 2^{Q}.

Upon an enqueue operation, the line index is extracted from the priority of the received packet tag to select the line where this incoming element should be inserted. This is done by a simple bit extraction of the Q most significant bits, to have the push index. For a dequeue, a priority encoder finds the pop index of the highest priority element in the exit buffer by selecting the first non empty line from the HPQS line counters, see Figure 4.3. Then, sorting in the hardware PQ, and storing the result back to the BRAMs are performed. The priority key that will reside inside each queue element can be further optimized to $P - Q + 1$ bits (the 1 bit addition is used to differentiate valid sort information from the invalid key represented by the maximum value over $P - Q + 1$ bits during hardware PQ sort). For example, for 64 queues, the priority key length stored inside each queue is only 11 bits. For 512 queues, the

priority key length stored is 8 bits. Thus, the metadata field varies from 21 bits up to 24 bits for the same example (i.e., for 64 and 512 distinct queues, respectively). The on-chip memory BRAM_18K can hold a maximum of 512 × 32-bit elements per memory block. It will be shown that during implementation (see Section 4.6.1), the width of the HPQS is varied from 64 to 1024 (1 Ki) elements, while QL is varied from 64 to 512.

A counter per queue line is needed during pop index calculation for the design to be fully pipelined, as the HLS tool fails to meet the 1 cycle target while accessing directly the exit buffer due to carried dependency constraint. This carried dependency is between the store operation of the top element in the exit buffer after each HPQS operation, and the load operation of the top element in each line for pop index calculation. To prevent this dependency, we access the counters during pop index calculation, while the exit buffer is used to pass the output and to store top elements only. In addition, all counters contain the actual stored number of elements per queue line. When reaching or exceeding the queue line capacity during an enqueue, the specific queue line counter is halted, while the last queued element is dropped. Normal counter operation is resumed once a dequeue is performed. More details on how to achieve best performances and matching handwritten designs through HLS are given in Section 4.5.

In this configuration, we propose two sorting types in the hardware PQ, a partial and full sort. Figure 4.4 depicts the partial sort (called P. sort) architecture. The hardware PQ is divided in groups, a group contains two packets representing the min and max elements. Each group is being connected with its adjacent groups, and each independently applying in parallel a common operation on its data. This hardware PQ architecture is register-based single-instruction–multiple-data (SIMD), with only local data interconnects, and a short broadcasted instruction. More details are provided in [9].

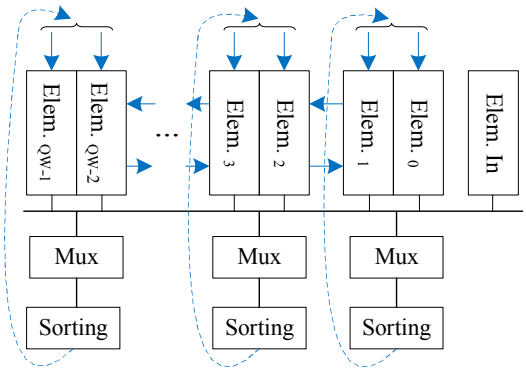


Figure 4.4 The hardware PQ architecture with partial sort.

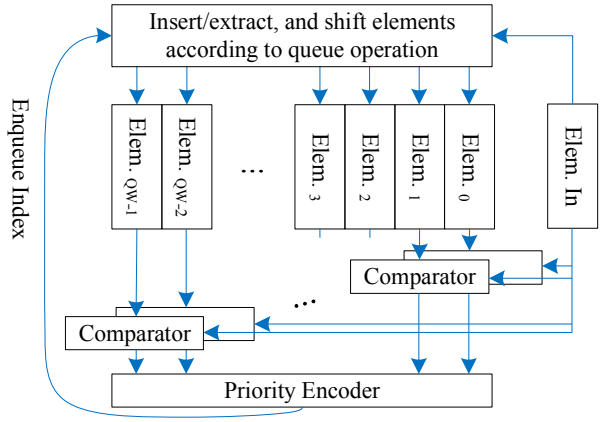


Figure 4.5 The hardware PQ architecture with full sort.

Figure 4.5 depicts the full sort (called F. sort) architecture. In here, the hardware PQ fully sorts the elements by comparing the incoming element to all existing elements during enqueue. The first activated comparator indicates the insertion location for the new element. This is found through a priority encoder leading to the appropriate enqueue index. A shift register is used to move the elements beyond the enqueue index by one location to the bottom of the hardware PQ. During a dequeue, the top element is removed from the hardware PQ, while the remaining elements are shifted to the top by one location.

From the performance analysis of the hardware PQ with partial sort [9], the performance decreases in $O(\log N)$, where N is the number of packets in each group, while the quality of dismissed elements when the queue is full is $1/N$ (lower is better). In this work, N is fixed to 2 packets in each group, for all queue sizes. However, in partial sort, the elements with similar priorities are not distinguishable in their order of departure according to their order of arrival. Full sort is proposed to guarantee the order of departure for such packets, as the new incoming elements are enqueued and placed after the existing onces. Also, this is suitable for some network equipment where out of order reception is not supported.

4.4.2 Type-2 Single-Queue Model

In this architecture, a high capacity PQ is proposed. In addition to enqueue and dequeue operations, a third basic operation which is the replace (simultaneous dequeue-enqueue) is supported. This architecture supports 16-bit priority and 48-bit metadata (64-bit elements) spread over virtual distinct priority queues, similar to type-1 QL, in here called virtual queue lines (VQLs). An enqueue is performed after receiving the line information of the first queue line with empty location through a priority encoder. This priority encoder selects the first

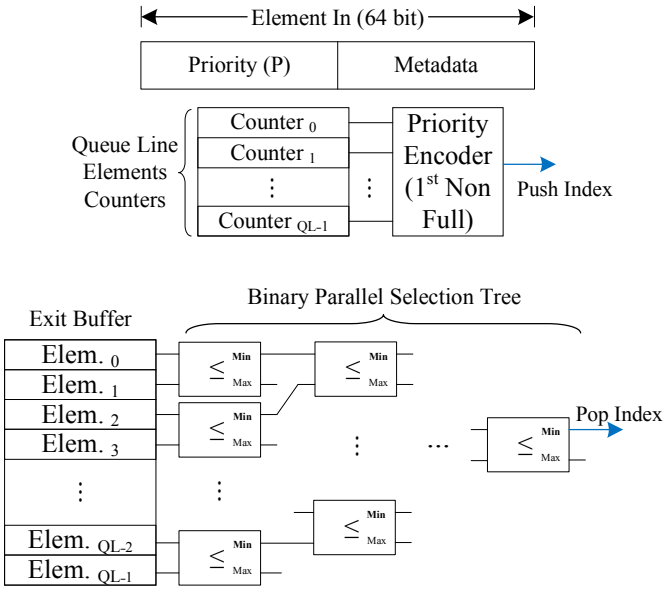


Figure 4.6 Push and pop indices calculation in HPQS type-2 architecture.

non full line counter as depicted in Figure 4.6. After sorting the upcoming element with the existing ones in the hardware PQ (load all elements from the BRAMs to the hardware PQ), the result is written to the same line in the BRAMs. In the case of a dequeue operation, a parallel comparison is made between the elements stored in the exit buffer with a binary parallel selection tree. The parallel selection tree has O(log n) time complexity, this complexity is almost constant during implementation, see Section 4.6.1. The exit buffer holds the highest priority element of each queue line. From the corresponding pop index, the content of on-chip memories are sorted to complete the dequeue operation the same way an enqueue operation is performed. Also, for this configuration, it will be shown that during implementation, the width of the HPQS is varied from 64 to 1024 elements, while VQL is varied from 64 to 512.

In this HPQS configuration, even if full sort is used, we cannot guarantee the order of departure according to the order of arrival for similar priority elements from the different queue lines. The selection of the minimum element to dequeue is done by a binary parallel selection tree, while the push index is selected by a priority encoder from any queue line with empty location. As we can enqueue and dequeue from any queue line, the order of departure for similar priority elements cannot be guaranteed. In the type-1 architecture, this order is guaranteed by the hardware PQ (with full sort) and the exit buffer. Each queue line contains the elements in ascending order of departure, and the selection in the exit buffer is done by a priority encoder that will choose the best element from top to bottom in priority order. This guarantees the order of departure for similar priority elements. Therefore, in the type-2 architecture only partial sort is used.

4.4.3 HPQS Functional Design

From a conceptual point of view, the HPQS is intended to work in a pipelined fashion. Each load/store from all BRAMs can be done in a single clock cycle. The HPQS operates as follows, in the first cycle (see circled numbers on Figure 4.2), according to the operation to perform either an enqueue or a dequeue in type-1, in addition to replace in type-2 architecture, an index is calculated. This index corresponds to the BRAMs line (together the respective lines of the parallel BRAMs contain 64 to 1 Ki elements, according to the queue capacity) to be loaded onto the hardware PQ. The ordering of the active queue is done in the second clock cycle, while a write back to the same BRAMs line to store the result is also done in the second clock cycle (More details are provided in Section 4.5.1).

It should be noted that each BRAM holds up to 512 categories, and each category can have from 64 up to 1 Ki elements. This leads to 512 queues with at most 1 Ki capacity. The load and store operations require two distinct buses of 1024 × 32-bit to transfer the elements in type-1, and 1024 × 64-bit elements in type-2 architectures. They are necessary to transfer the stored elements in the BRAMs to the hardware PQ and vice versa. When the HQPS is generated with its full capacity, 2^{16} and 2^{17} nets are instantiated for each configuration, respectively. This impacts performance (more details are given in the implementation results Section 4.6).

4.5 HLS Design Methodology and Considerations

In this section, we first present the analysis of operations required by the proposed HPQS design. Then, we detail the steps applied in HLS to obtain the desired throughput and latency.

4.5.1 Analysis of HPQS Operations

The timing diagram demonstrating correct operation of the proposed HPQS is shown in Figure 4.7. The required operations for the HPQS are to extract (in type-1 architecture) or choose (in type-2 architecture) the line to enqueue, dequeue, or replace an element, load the line content from the BRAMs to the hardware PQ for sorting, and finally write back the result to the same line in the BRAMs. Therefore, the HPQS operations consist in reading the storage memory (steps C0-C1), sorting the queue elements (step C1), and writing back the result to the same memory location (step C1). These are the specific tasks done by the proposed HPQS for each queue operation at any given clock cycle.

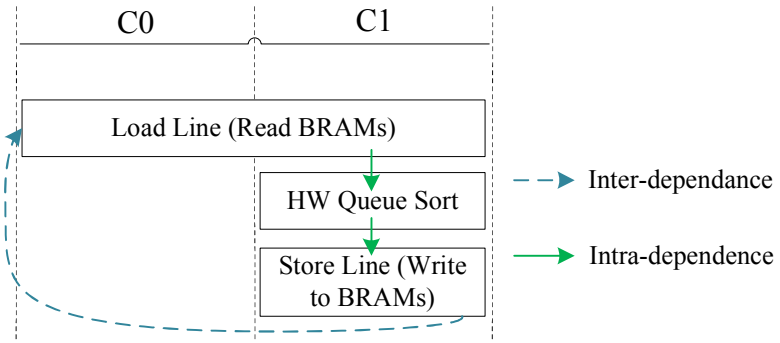


Figure 4.7 Proposed HPQS pipeline operations timing diagram.

4.5.2 Design Methodology

High-level synthesis enables raising the design abstraction level, while providing more flexibility by automatically generating synthesizable Register Transfer Logic (RTL) from C/C++ models, as compared to Hardware Description Language (HDL) hand-written designs. Also, HLS requires less design effort, when performing a broad design space exploration, such that many derivative designs can be obtained with a small incremental effort. In addition, design space exploration can be performed through different available directives and constraints provided by the tool. Using directives, the user can guide the HLS tool during C-synthesis. Thus the designer can focus on the algorithmic design aspects, rather than on low-level details required when using HDL. The main metrics used to measure performance in HLS designs are area, latency, and Initiation Interval (II). In this work, we performed all experiments with Vivado HLS while the design is coded in C++. Appropriate design iterations were applied to refine the HPQS (code optimization and enhancement). The code was thoroughly tested. Finally, design implementation metrics are defined such as the target resource usage, desired throughput, clock frequency, and design latency.

We performed a thorough design space exploration for the HPQS through HLS targeting minimum latency, equivalent memory usage (number of BRAMs) and highest throughput. The total considered HPQS capacity is 512 Ki. From Figure 4.7, it can be seen that the minimum latency that can be achieved from our design operation, and initiation interval (II) are 1 clock cycle each, i.e., every clock cycle an output packet tag is ready. To target this optimal performance through HLS, the three directives that we focused on are: a latency directive targeting 1 clock cycle, a pipeline directive with II of 1 clock cycle, and a memory dependency directive asking for separate true dual port memories for accessing the element information through a read and/or write in the same cycle in the BRAMs. As HLS constraint,

we target the lowest feasible clock period without violating the desired design latency and II mentioned above.

The partition directive was used to guide the tool to use only logic resources to implement logic and not the BRAMs available in the FPGA, to reduce design latency by cutting down the memory access time for the hardware PQ. To map the storage to the on-chip BRAMs, the resource directive is used with the option "true dual port RAM" enabling load/store in the same cycle. The pipeline directive is used to target an II of 1 clock cycle. All the mentioned directives are used together to generate the HPQS design. It should be noted that a bypass is not required for back-to-back similar line accesses in the BRAMs as the previous line content is already in the hardware PQ. In here, the load operation from the BRAMs is simply discarded (see intra-dependences in Figure 4.7). More details on the experimental results of placement and routing in FPGA for different HPQS configurations and capacities are provided in Section 4.6.1.

4.6 Implementation Results

In this section, we detail the hardware implementation of our proposed HPQS architecture, resource usage and achieved performance, for different configurations (type-1 and 2) and capacities. Then, comparisons to existing works in the literature are discussed.

4.6.1 Placement and Routing Results

The proposed HPQS was implemented on a Xilinx Zynq-7000 ZC706 board (based on the xc7z045ffg900-2 FPGA) and on a XCVU440 Virtex UltraScale device (xcvu440-flgb2377-3-e). The type-1 architecture implementation results are summarized in Figure 4.8, under ZC706 on the left column (Figure 4.8a to 4.8d), and the XCVU440 results are summarized in the right column (Figure 4.8e to 4.8h). The resource utilization in terms of the number of look-up tables (LUTs) are reported in Figure 4.8a, and of the number of flip-flops (FFs) in Figure 4.8b. For performance, we report the achieved clock period in Figure 4.8c. Also, the dynamic power consumption of the proposed HPQS type-1 architecture are depicted in Figure 4.8d. In the same order, implementation results when targeting a XCVU440 device are reported in Figure 4.8e–4.8h, respectively. In addition, the reported results are in terms of the HPQS queue width (QW), sort types (see the legends: partial sort labeled P. sort, and full sort labeled F. sort), and the number of queue lines (QLs).

The type-2 architecture implementation results are reported in Figure 4.9 in the same manner we reported the type-1 architecture implementation results. In here, the reported results are

in terms of supported queue operations (with and without simultaneous dequeue-enqueue or replace), and the number of virtual queue lines (VQLs). Moreover, in the type-2 architecture, under ZC706, various HPQS queue widths are explored from 64 up to 512 elements. By contrast, the queue widths range from 64 to 1024 when targeting the XCVU440 device. These ranges are determined by the number of BRAM_18K available in the FPGA devices. The ZC706 has only 1090 blocks, while the XCVU440 has 5040 blocks. Recall that a BRAM_18K can hold an entire column of the HPQS storage with 512 $\times$ 32-bit elements. In the type-2 architecture, each element is 64-bit that consumes 2 BRAMs to support the full width of each entry. It should be noted that the achieved throughput and latency for all HPQS configurations are 1 clock cycle.

In what follows, we discuss in details the obtained HPQS implementation results under both FPGA devices in terms of LUTs, FFs, achieved clock period and dynamic power consumption.

The resource consumption of the different HPQS configurations can be divided into four main parts: hardware PQ, exit buffer, storage resource, and counters of elements per queue line. The hardware PQ depth (QW) is varied from 64 to 1024 elements, while the maximum HPQS height is 512. In the hardware PQ implementation, only FFs and LUTs were used to obtain a fast pipelined architecture achieving 1 clock cycle per queue operation. The exit buffer was implemented as a register-based array to hold only the top elements of each queue line of the HPQS. The line counters are used to break up the dependency on checking if the queue line is empty/full at each queue operation on the exit buffer. Without these line counters, the HLS tool was not able to pipeline the design to 1 clock cycle. The BRAM_18K usage was found to reflect directly the width of the HPQS (QW), as each on-chip memory is mapped to hold an entire column of the proposed HPQS storage (see Figure 4.2).

In type-1 configuration (see Figure 4.8a and 4.8e) with both FPGA devices (ZC706 and the XCVU440, respectively), increasing the capacity of the hardware PQ from 64 to 1024 elements with partial sort, the LUTs consumption linearly increases as more groups are attached in the SIMD hardware PQ. However, with full sort, this increase is linear over a range of capacity between 512-640 elements, then it stabilizes between 768-896 after which it increases again. This is the complexity of full sort with the priority encoder for index selection through an array of comparators (see Section 4.4.1). When increasing the number of queue lines or the height of HPQS, from 64 to 256 lines with queue width not exceeding 512 elements, the LUTs usage is quite similar with both sort types (see Figure 4.8a and 4.8e). Beyond 256 queue lines and 512 elements in HPQS width, the increase in the number of LUTs is no longer linear. The more lines in the HPQS, the more complex is the decoder of line index and its routing (width of the multiplexers), this complexity is logarithmic and appear to follow a stair case

function. The same tendency can be seen for both devices (Zynq-7 and Virtex Ultrascale), where they have similar slice organization with 6-input LUTs. It should be noted that the XCVU440 device has more resources (11.5× more in LUTs/FFs) compared to the ZC706 FPGA.

Type-2 architecture has a selection tree used during dequeue, and a priority encoder for line selection during enqueue in addition to the resource of type-1 architecture, with partial sort is used in hardware PQ. The LUTs usage (see Figure 4.9a and 4.9e) is higher with similar tendency (linearly increasing for 64 up to 256 queue lines or HPQS height) and for different hardware PQ widths (64 up to 1024 elements). Also, supporting the third queue operation, i.e. replace, does increase the resource consumption by 37% in the worst case with the ZC706 FPGA, and 54% with the XCVU440 device.

Regarding the FFs usage (see Figure 4.8b, 4.8f for the type-1 architecture, and Figure 4.9b, 4.9f for the type-2 architecture under ZC706 FPGA and XCVU440 device, respectively), it reflects directly the use of memory resource of the hardware PQ, exit buffer and the line counters in all HPQS configurations. For example, the hardware PQ uses element length × queue width, for the counters 11-bit × queue lines, and the exit buffer element length × queue lines. let us recall that the element length in the type-1 architecture is 32-bit, while in the type-2 architecture is 64-bit. Also, as the HPQS capacity increases, all the above FFs resource relations are linear with the height and width dimensions.

For the performance metrics, the clock period achieved with the type-1 architecture (see Figure 4.8c, 4.8g) with partial sort is better compared to the type-2 architecture (see Figure 4.9c, 4.9g). It should be mentioned that the type-1 architecture is intended only for strict priority scheduling with distinct queues. If used as a priority queue, proper priorities repartition is advised. Type-2 is a high capacity priority queue that supports by default type-1 functionality. For type-1 with full sort, for both devices (see Figure 4.8c, 4.8g), the performance decreases beyond 512 hardware PQ elements capacity with lower performances compared to partial sort. However, the achieved clock period is more stable and almost constant with partial sort in both FPGA devices. The XCVU440 device achieved better results than the Zynq-7. This is mainly due to the fact that the XCVU440 device is less prone to net congestion during routing, as it has more resources (11.5×) compared to the ZC706 FPGA that used up to 99.0% of its LUTs, as reported in Table 4.1 for the largest design. It should be noted that for a clock period of 16.8 ns, the different designs are capable of supporting links up to 40 Gb/s for 84 bytes minimum size Ethernet packets (including minimum size packet of 64 bytes, preamble and interpacket gap of 20 bytes).

For dynamic power consumption, the XCVU440 and ZC706 FPGAs have a similar tendency of linear growth with the HPQS capacity (for type-1 with partial sort and type-2 configurations, see Figure 4.8d, 4.8h for the type-1 architecture, and Figure 4.9d, 4.9h for the type-2 architecture under ZC706 FPGA and XCVU440 device, respectively). It should be noted that with smaller designs, the lower the achieved clock period, the more power is consumed. This can be seen with designs having up to 256 lines with hardware PQ width up to 640 elements (in type-1 full sort, see Figure 4.8d, 4.8h). Inversely, the larger is the design (beyond 256 queue lines), the higher is the clock period leading to lower power consumption. Overall, in type-1 configuration, both FPGA devices have similar tendency. With the type-2 architecture, the only difference is in the largest designs (HPQS height of 512, see Figure 4.9d, 4.9h), the ZC706 is nearly fully used leading to lower clock period and dynamic power consumption compared to the XCVU440.

Table 4.1 Percentage of resource utilization for the largest designs of HPQS in ZC706 and XCVU440 FPGAs

Board / Device	HPQS Type	Design (Height × Width)	Slice/ CLB (%)	BRAM (%)	Clock Period (ns)	Dynamic Power (W)
ZC706 Zynq-7000	1	Partial sort 512×1024	88.4	94.0	19.75	1.55
	1	Full sort 512×1024	99.8	94.0	65.25	0.71
	2	w/o replace 512×512	98.6	94.0	37.13	1.28
	2	w/ replace 512×512	99.0	94.0	37.17	1.27
XCVU440 Virtex UltraScale	1	Partial sort 512×1024	9.2	20.3	18.62	2.08
	1	Full sort 512×1024	15.9	20.3	38.75	2.51
	2	w/o replace 512×1024	16.4	40.6	22.62	6.28
	2	w/ replace 512×1024	18.5	40.6	25.94	6.52

Table 4.1 depicts the percentage of the FPGA resource usage (slice under ZC706, configurable logic block (CLB) under XCVU440, and BRAM memory) after HPQS placement and routing of the largest designs (type-1 and 2), with the achieved clock period, and dynamic power consumed in both FPGA devices. Note that in the ZC706 FPGA, we used 88 to 99% of available slices, and 94.0% of BRAMs, that led to lower performance in comparison with the

ZC706 XCVU440

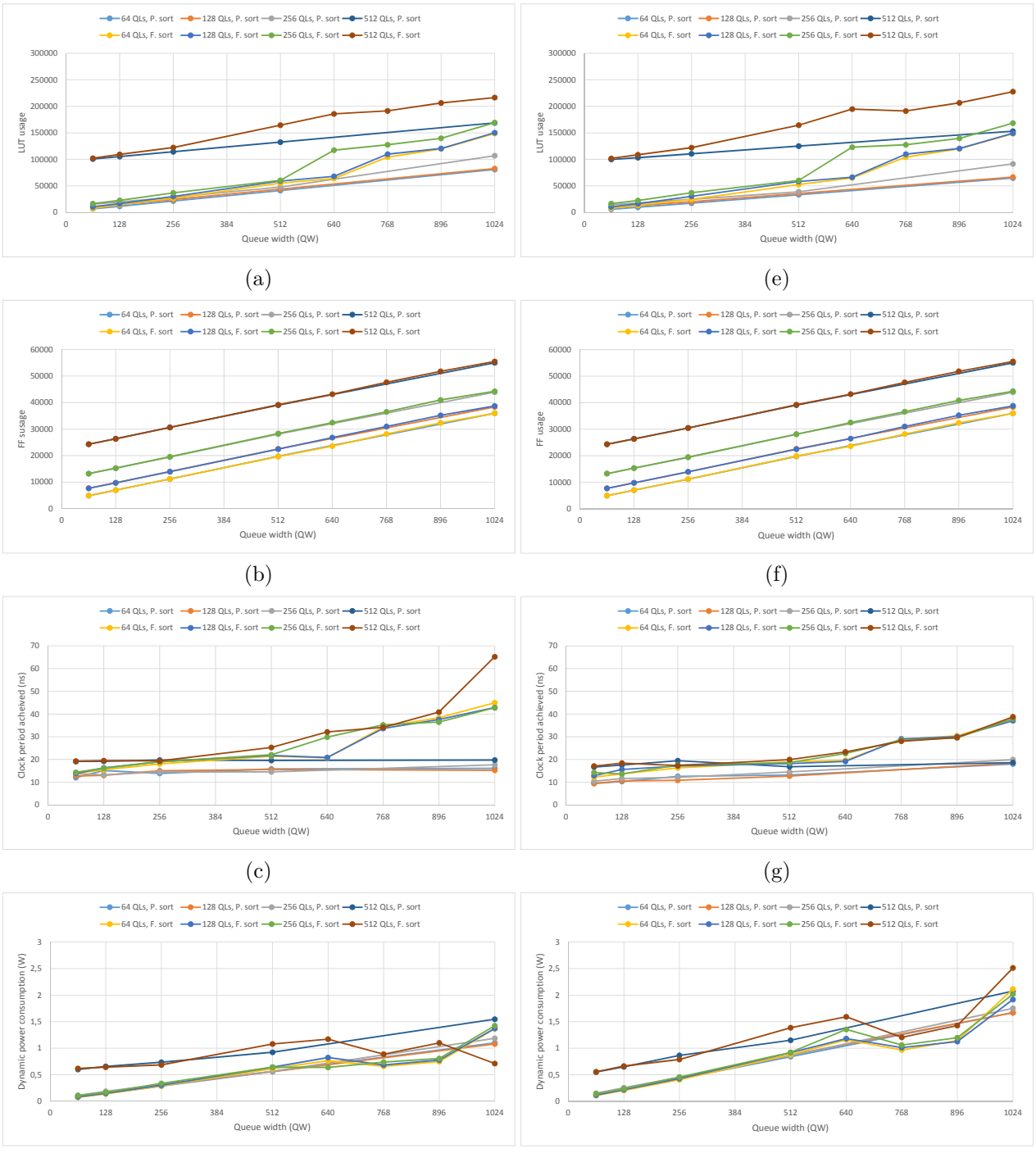


Figure 4.8 The HPQS type-1 configuration implementation results for 32-bit element on the ZC706, and XCVU440 FPGAs with: (a) LUT usage, (b) FF usage, (c) achieved clock period, and (d) dynamic power consumption under ZC706. Similarly from (e-h) under XCVU440.

ZC706 XCVU440

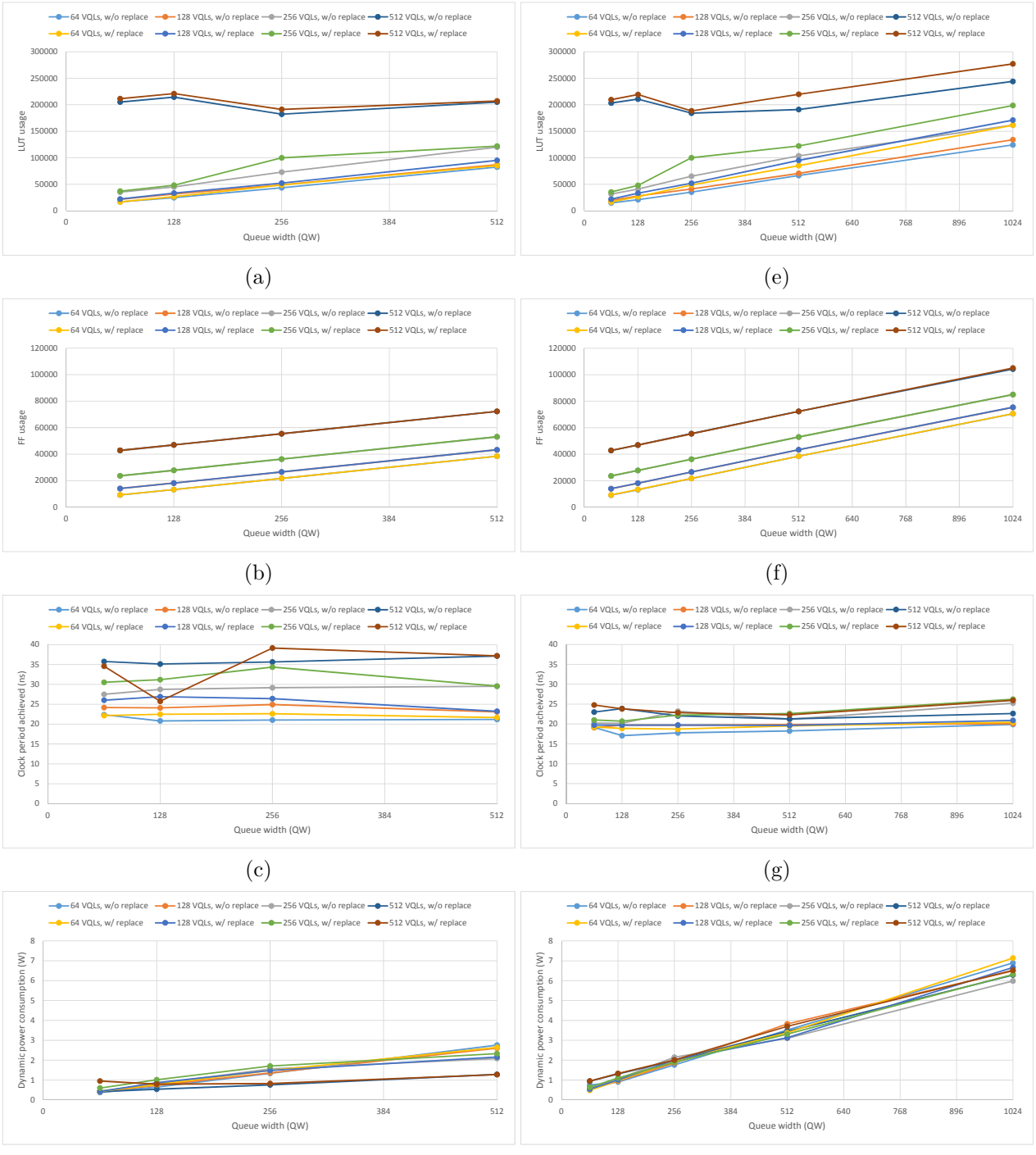


Figure 4.9 The HPQS type-2 configuration implementation results for 64-bit element on the ZC706, and XCVU440 FPGAs with: (a) LUT usage, (b) FF usage, (c) achieved clock period, and (d) dynamic power consumption under ZC706. Similarly from (e-h) under XCVU440.

Table 4.2 Performance comparison for different priority queue architectures

Architecture	Queue Depth	# Cycles per Hold Operation	Clock (ns)	Priority ; Metadata / Platform	Throughput (Mpps)
Shift register [57]	1 Ki	2 (no replace)	20.0	16 ; NA / ASIC	25.0
Systolic array [57]	1 Ki	2 (no replace)	22.2	16 ; NA / ASIC	22.5
Shift register [27]	1 Ki 2 Ki	2 (no replace)	4.92 6.64	16 ; 32 / FPGA (Cyclone II)	102 75.3
Hybrid register/binary heap [39]	1 Ki	4 : replace 11 : others	~7.5	13 ; 51 / FPGA (Zynq-7)	33.3 12.1
	2 Ki	2 : replace 11 : others	~8.0		62.5 11.4
	4 Ki	1 : replace 11 : others	13.1		76.3 6.9
	8 Ki	1 : replace 11 : others	15.0		66.7 6.1
Pipelined Heap [41]	16 Ki	2 - 17 (no replace)	5.56	18 ; 14 / ASIC	Best: 90 Worst: 10.6
PIFO [72]	1 Ki	2 : replace 4 : others	13.9	16 ; 32 / FPGA (Zynq-7)	36.0 18.0
Lookup tree (trie)[53]	8 Ki	4: replace 8: others	7.0	NA ; 12 / ASIC	35.8 17.9
Multi-bit tree[75]	8 Ki	4 (no replace)	4.63	NA ; 12 / FPGA (Virtex II)	54
Hardware PQ [9]	1 Ki	2 : no replace 1 : replace	3.31 4.0	16 ; 32 / FPGA (Zynq-7)	151 250
Proposed HPQS Type-2	64 Ki	2 : no replace 1 : replace	23.08 23.22	16 ; 48 / FPGA (Zynq-7)	21.7 43.1
	256 Ki	2 : no replace 1 : replace	37.13 37.17		13.5 26.9
	64 Ki	2 : no replace 1 : replace	19.87 19.64	16 ; 48 / FPGA (XCVU440)	25.2 50.9
	512 Ki	2 : no replace 1 : replace	22.62 25.94		22.1 38.5

XCVU440 UltraScale device. The largest design (512 Ki capacity with type-2) consumes less than 20% of the CLBs and 40.6% of the BRAMs resources when targeting the XCVU440 device. So, the unused resources could be easily exploited to scale the HPQS design to even larger capacity beyond the proposed 512 Ki elements by 4.0×, and 2.0× for the type-1 and 2 HPQS architectures with 1024 elements queue width, respectively. Note that when targeting the ZC706, the largest HPQS type-2 design implemented was 256 Ki elements in capacity due to limited number of BRAMs, leading to 94.0% memory usage.

4.6.2 Comparison With Related Works

The proposed HPQS supports enqueue and dequeue operations for type-1, in addition to replace in type-2 configuration. The number of cycles between successive dequeue–enqueue (hold) operations is 2 clock cycles, and only 1 clock cycle when replace is supported, as reported in Table 4.2. Indeed, each queue operation takes 1 cycle to finish. This is less than the binary heap [39] and p-heap architectures [41]. The reported shift register and systolic architectures in Moon's work [57] have a latency of 2 clock cycles for en/dequeue. In case of the shift register proposed by Chandra [27], the performance degrades logarithmically. Compared to the p-heap architecture [41], even though it accepts pipelined operations each clock cycle (except in case of successive deletions), the latency is O(log n) in terms of the queue capacity, against O(1) time latency for our proposed HPQS architecture.

Even though the hardware PQ is fast, achieving 3.31 ns per operation with only enqueue/dequeue, and 4.0 ns with replace (see Table 4.2), the BRAMs distribution in the FPGA span many columns. During placement and routing of the largest HPQS designs (up to 512 lines × 1024 elements), long net delays tend to be generated by the hardware PQ to BRAMs connections (recall that the full architecture requires 2^{16} in type-1 and 2^{17} nets to connect the BRAMs to the hardware PQ as explained in Section 4.4.3). This impacts directly the overall performance of the design as the clock period of the whole HPQS is 23.0 and 19.7 ns for the ZC706 and XCVU440 FPGA devices respectively with 64 Ki capacity (type-2). For 256 Ki capacity with ZC706 (largest routed design under this FPGA), it decreases to 37.0 ns. With 512 Ki design with XCVU440 device, the clock period decreases to 22.6 and 25.9 ns w/o and w/ support of replace operation, respectively. This design supports ½ million elements in a single FPGA, against a few thousands in previously published works.

In our proposed HPQS, we achieved a guaranteed performance and latency due to the fixed number of cycles (O(1) complexity). This constant number of cycles is independent of the hardware PQ width and the HPQS capacity, unlike the O(log n) time for the dequeue operation observed with the heap [39, 41, 87], where n is the number of nodes (keys). The through-

put achieved with the proposed solution is 22.1 million packets per second (Mpps) without replace, and 38.5 Mpps with replace for 512 Ki total capacity (type-2) under XCVU440 device. From works reported in Table 4.2, only some queues with 2 Ki and less capacity have throughputs better than our proposed HPQS. Beyond this capacity, either the designs have problem fitting in the targeted FPGA like in [27, 57, 72], or the throughput degrades bellow our achieved throughputs [39, 41]. Compared to [1, 15, 39, 57, 86], the reported throughput of the HPQS is independent of the queue capacity.

4.7 Conclusion

This paper proposed and evaluated a hybrid priority queue architecture intended to support the requirements of todays high-speed networking devices. The proposed HPQS was coded in C++ and synthesized using Vivado HLS. The first HPQS configuration with distinct-queues model is intended for strict priority scheduling. The second configuration is intended to offer a single large capacity priority queue for sorting purposes.

The proposed HPQS can support pipelined operations, with one operation completed at each clock cycle, with a capacity up to ½ million elements in a single FPGA. Also, the achieved throughput is comparable to similar related works in the literature, while supporting 10 to 40 Gb/s links. The achieved latency is in O(1) time complexity for the different queue operations independent to the total number of packets tags or HPQS capacity.

CHAPTER 5 ARTICLE 3: A HIGH-SPEED, SCALABLE, AND PROGRAMMABLE TRAFFIC MANAGER ARCHITECTURE FOR FLOW-BASED NETWORKING

This chapter contains the paper entitled "A High-Speed, Scalable, and Programmable Traffic Manager Architecture for Flow-Based Networking" published in the IEEE Access[1] journal. The main idea that led to this paper was to propose means to perform data plane traffic management in the context of high-speed networking devices addressing today's networking requirements of low latency and high throughput. A programmable and scalable flow-based traffic manager is proposed. The design provides the core functionalities of traffic management that are policing, scheduling, shaping and queue management. The SIMD hardware PQ (see chapter 3) is used to sort out the scheduling time of packets, that is crucial to keep this traffic scheduling at gigabit link rates. This traffic manager was implemented using HLS, and was validated on the ZC706 FPGA board.

Note that for those who read chapter 2, Section 2.3, can skip Section 5.2.3 of this chapter.

Abstract—In this paper, we present a programmable and scalable traffic manager (TM) architecture, targeting requirements of high-speed networking devices, especially in the software defined networking (SDN) context. This TM is intended to ease deployability of new architectures through field-programmable gate array (FPGA) platforms, and to make the data plane programmable and scalable. Flow-based networking allows treating traffic in terms of flows rather than as a simple aggregation of individual packets, which simplifies scheduling and bandwidth allocation for each flow. Programmability brings agility, flexibility, and rapid adaptation to changes, allowing to meet network requirements in real-time. Traffic management with fast queuing and reduced latency plays an important role to support the upcoming 5G cellular communication technology. The proposed TM architecture is coded in C++, and is synthesized with the Vivado High-Level Synthesis (HLS) tool. This TM is capable of supporting links operating beyond 40 Gb/s, on the ZC706 board and XCVU440-FLGB2377-3-E FPGA device from Xilinx, while achieving 80 Gb/s and 100 Gb/s throughput, respectively. The resulting placed and routed design was tested on the ZC706 board with its embedded ARM processor controlling table updates.

[1]I. Benacer, F.-R. Boyer, and Y. Savaria, "A High-Speed, Scalable, and Programmable Traffic Manager Architecture for Flow-Based Networking," in IEEE Access, vol. 7, pp. 2231-2243, 2019, © 2019 IEEE. This work was supported in part by the Natural Sciences and Engineering Research Council of Canada (NSERC), in part by Prompt Québec, in part by Ericsson Research Canada, in part by Mitacs, and in part by Kaloom.

5.1 Introduction

With the growing demand for higher network bandwidth and need to satisfy various subscribers requirements for a wide range of connected devices running applications such as smart phones, watches, detectors, etc., network operators and service providers are consistently upgrading their equipment. Many advanced applications of the so-called 5G [37, 62] next generation communication infrastructures impose requirements for very low latency packet switching and short delay routing.

With the current thrust toward Software Defined Networking (SDN) [85], it becomes natural to associate each packet to a flow. A crude definition of a flow is a set of packets associated with a client of the infrastructure provider having the same header information or sharing common specific packet fields. For instance, a flow could correspond to a web page, an email, a file, or a streaming application (video, call or conference), etc. In cellular networks like 5G, bandwidth is assigned to subscribers, so each packet is already part of a flow with some assigned bandwidth. For that reason, one of the feasible solutions is to tag the incoming packets with flow numbers as soon as they enter the network. This helps allocating bandwidth and simplifies scheduling.

Flow tagging is expected to become part of the context of next generation networking equipment as part of the so-called flow-based networking [85]. In the literature, priority queues (PQs) have been used to maintain real-time sorting of queue elements at link speeds [1, 9, 12, 39, 57, 87]. Also, different schemes and data structures were presented to deal with such networking needs and to maintain this priority based scheduling in today's high-speed networking devices.

Programmable network capability brings and improves agility, while enabling real-time prioritization of heavy traffic such as video during special events for example. An automatically tuned network is more flexible than one that would be subject to manual tuning by a network operator. Automatic control loops could orchestrate the tuning of the network to meet the needs of each application on the fly. For a flow-based traffic manager (TM), it sets a need for various features. For instance, bandwidth that is not used by some flows could be dynamically re-allocated to more active ones. Specific flows or applications could also be dynamically prioritized, i.e., video call over file sharing, etc. [6, 47, 66].

Traffic managers are usually located in a line card, providing the necessary queuing and scheduling functionalities for the incoming traffic in network processing units (NPUs) [40, 60, 86]. Packet scheduling is a demanding task dealing with priorities that are implicit and that depend on several factors, for example, protocols, traffic intensity, congestion, etc.

Usually, packet classification precede the TM. Commercially available TM solutions are either rigid because they rely on ASIC implementations, or require high-speed processing platforms [3, 21, 40]. In the research community, especially academia, only a few published works report complete TM architectures [7, 8, 60], while the majority of previous publications focus on specific functionalities such as scheduling [12, 28, 44, 54], and congestion management [34].

In this work, we claim the following contributions:

1. An FPGA-prototyped TM architecture offering programmability, scalability, low-latency with scheduling packet departures in a constant 2-cycle per packet. This TM architecture exploits pipelined operations, and supports links operating beyond 40 Gb/s without loosing performance during flow updates (tuning), with minimum 64 byte sized packets.

2. The TM integrates core functionalities of policing, scheduling, shaping, and queue management for flow-based networking entirely coded in C++. High-level synthesis provides more flexibility, and faster design space exploration by raising the level of abstraction.

3. TM programmability can be supported with the popular P4 (programming protocol-independent packet processors) language, together with TM integration as a C++ `extern` function.

Even though the reported TM architecture was validated with an FPGA platform, it could also be synthesized as an application-specific integrated circuit (ASIC), since our configurability is not obtained through re-synthesis. Of course, further flexibility and configurability can be supported on FPGA if a pass through the tool chain from high-level synthesis to routing is allowed, but such configuration is not currently supported on the fly.

The remainder of this paper is organized as follows. In Section 5.2, we present a literature review of some existing TM solutions. In Section 5.3, we describe the architecture of the TM, its underlying modules and some supported scheduling schemes. In Section 5.4, we present two solutions to deal with schedule time overflow. In Section 5.5, we present the HLS methodology and directives / constraints used to achieve the desired performances. In Section 5.6, hardware implementation of the proposed architecture and comparisons to other works in the literature are provided, while Section 5.7 summarizes our main conclusions.

5.2 Related Work

In this section, we first introduce flow-based networking, programmable switches, and then review relevant works related to traffic management for different platforms.

5.2.1 Flow-based Networking

The core idea behind flow-based networking is to process the network traffic in terms of flows rather than individual packets. An early design of a flow-based networking device is the Apeiro router from Caspian, in which a flow is defined as a set of packets sharing the same header characteristics or mainly the 5-tuple (source and destination IP, source and destination port, and protocol). The Apeiro flow-based router ensures quality of service (QoS) of each flow and fairness versus other traffic types [66].

Software defined networking enables the separation of the control of network devices from the data they transport, and the switching software from the actual forwarding network. In other terms, the control plane is separated from the data plane. OpenFlow is a standard defined by the Open Networking Foundation (ONF) for implementing SDN in networking equipment. This protocol allows the OpenFlow controller to instruct an OpenFlow switch on how to handle incoming data packets. These control functions (control actions) are structured as flows. Each individual flow contains a set of rules for filtering purposes. The flow actions, i.e., forward, drop, modify, etc., and statistics gathering are grouped in the flow table. The OpenFlow architecture enables flow-based networking with capabilities including software-based traffic analysis, centralized control, dynamic updating of forwarding rules, etc. [47].

5.2.2 Programmable Switches

In the literature, works around hardware programmable switch architectures [5] and other about their software abstractions [17] were proposed. While many packet-processing tasks can be programmed on these switches, traffic management is not one of them (more details are given in the next subsection). Programmable switches can benefit from our proposed TM by the use of `extern`s through P4 language in its latest release $P4_{16}$. From architectural point of view, the TM is seen like an external accelerator attached to the switch pipeline providing the necessary TM functionality and programmability needed in today's networks (More details are provided in Section 5.3.2.6c).

5.2.3 Traffic Managers

Traditionally, traffic management has been implemented using hardwired state machines [73]. It evolved from dedicated modules in NPUs [2, 32] to separate standalone solutions [21, 40, 79] that can be used as co-processors. Generally, TMs are considered as independent processing elements attached to a flexible pipeline in a NPU. Current solutions use dedicated traffic management integrated within NPUs to speed-up traffic processing, with external memories for packet buffering and queuing purposes. A TM can be found in the data path of a NPU, of a line card, etc. This corresponds to the so-called flow-through mode. By contrast, in the look-aside mode, the TM is outside the data path and it communicates only with the NPU or the packet processor, acting as a co-processor (see Figure 5.1). The NPU sends tags, temporary headers, or packet descriptors to the TM. The packet buffer is only attached to the packet processor.

The available traffic management solutions in the literature are essentially commercial products with only few works done in academia. Paulin [64] proposed a multiprocessor system-on-chip (MP-SoC) architecture for traffic management of IPv4 forwarding. The proposed platform is composed of multiple configurable hardware multi-threaded processors, with each processor running part of the traffic management features or tasks. To process more traffic and to cope with network requirements, this architecture requires more processors, eventually limiting its scalability.

Zhang [86] proposed a complete TM implemented in an FPGA platform, focusing on the programmability and scalability of the architecture to address today's networking requirements. However, the queue management solution that was adopted slows down the entire system with at least 9 cycles per enqueue/dequeue action, and an implementation running at 133 MHz. This TM solution achieved around 8 Gb/s for minimum size 64 byte packets.

Khan [42] proposed a traffic management solution implemented with dedicated circuits that can support 5 Gb/s with full duplex capabilities. Khan showed all the design steps up to the physical realization of a TM circuit. This solution remains rigid as it targets an ASIC. This design choice limits its ability to support future networking needs.

Table 5.1 summarizes the TM solutions offered by commercial vendors and published by academia, along with the platform for which they were developed, their configuration and the reported throughput.

Table 5.1 Traffic management solutions

Company / Researcher	Platform	Configuration	Throughput (Gb/s)
Zhang (2012) [86]	FPGA (Virtex-5)	Look-aside	8
Altera (2005) [40]	FPGA (Stratix II)	Flow-through	10
Broadcom (2012) [21]	ASIC	Flow-through	Up to 200
Mellanox (Ezchip) (2015) [32]	ASIC	—	Up to 400
Agere (2002) [2]	ASIC	Flow-through	10
Xilinx (2006) [79]	FPGA (Virtex-4)	Look-aside	—
Paulin (2006) [64]	MP-SoC (90 nm)	Look-aside	2.5
Khan (2003) [42]	ASIC (150 nm)	Look-aside	5
Bay (2007) [3]	ASIC (110 nm)	Flow-through	50 or 12×4

5.3 Traffic Manager Architecture

In this section, we present a generic TM architecture and its functionalities in a line card. Then, we detail its underlying modules and some supported packet scheduling schemes.

It is of interest to mention that this work is an extension of a previous related work [7, 8], which is extended as follows:

1. Integration of a queue manager (QM) with throughput reaching 100 Gb/s for minimum sized packets, which is a significant improvement over the previously reported 47 Gb/s (More details are given in Section 5.3.2.5).

2. Policer functionality with decision based on actual queue occupancy and flow heuristics to actively assess the flow state and manage any eventual congestions and flow attacks (see Section 5.3.2.2).

3. Analysis of TM operations leading to improvements that allowed matching the performance of hand-written register transfer logic (RTL) codes from an HLS design (see Section 5.5.1).

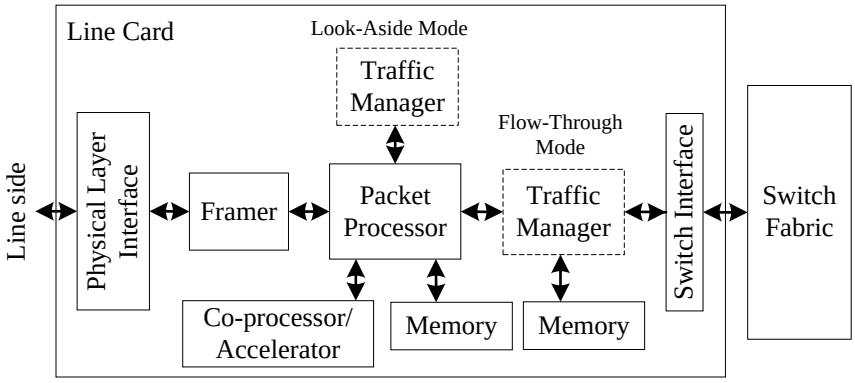


Figure 5.1 Generic architecture around the traffic manager in a line card.

4. Test and validation on the ZC706 FPGA board of the TM design to verify its correct functionality after placement and routing (More details are given in Section 5.6.2).

5.3.1 Traffic Manager Overview and Functionalities

Traffic management allows bandwidth management, prioritizing and regulating the outgoing traffic through the enforcement of service level agreements (SLAs). A SLA defines the requirements that a network must meet for a specified customer, some service, or level of service that must be ensured to a subscriber by the service provider. Popular level of service measures include guaranteed bandwidth, end-to-end delay, and jitter.

Traffic management is applied to different types of traffic that have distinct characteristics and requirements to meet. For example, traffic characteristics are the flow rate, flow size, burstiness of the flow, etc. while traffic requirements are the QoS in general. Overall, network operators are targeting to meet all SLAs, to achieve fairness and enforce isolation, while prioritizing the different traffic flows, and to maximize network utilization through traffic management.

A generic TM in a line card (switches, routers, etc.) is depicted in Figure 5.1. The packet processor classifies to a specific flow the data traffic prior entry into the TM. The classified data traffic allows the TM to prioritize and decide how packets should be scheduled, i.e., when packets should be sent to the switch fabric. Traffic scheduling ensures that each port and each class of service (CoS) gets its fair share of bandwidth. Traffic should be shaped before being sent onto the network. The shaper enforces packets to follow a specific network pattern by adding delays. The shaper provides such delays to outgoing traffic to ensure it

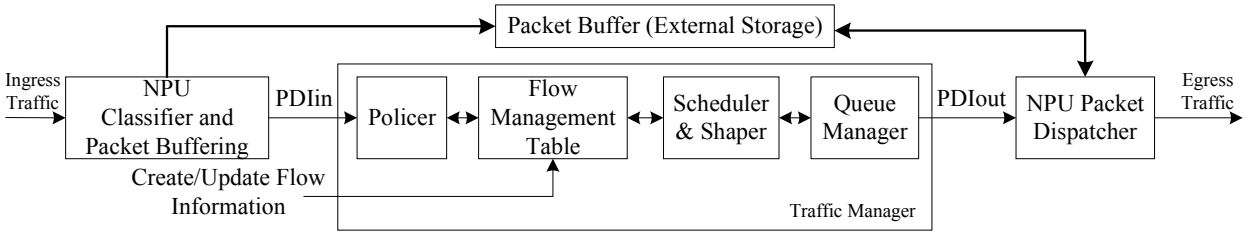


Figure 5.2 Proposed traffic manager architecture.

fits a specific profile (link usage, bandwidth, etc.) and enforces QoS. Packet congestion can cause severe network problems, including throughput degradation, increased delay, and high packet loss rates. Congestion management can improve network congestion by intelligently dropping packets. The policer makes decisions to drop packets preventing queues overflow and network congestion.

5.3.2 Traffic Manager Structural Design

The proposed TM architecture is depicted in Figure 5.2. It is composed of the following modules: policer, flow management table (FMT), scheduler, shaper, and a QM. The system architecture assumes that each packet received by the TM has already been tagged with a flow number by the packet processor as per the classification stage. The TM handles only the packet descriptor identifier (PDI). Each PDI contains the priority, packet size, flow ID or number, and its address location in the packet buffer. The PDI may contain other attributes. The size of PDI fields are determined according to the system configuration. For example, to support any standard Internet packet size, the PDI size field is set to 16 bits. The PDI enables packets to be located in the network, providing fast queue management with reduced buffering delays, where the entire packet is stored outside the TM. Usually packet buffering is handled by the NPU. Using the PDI's has the same impact as if real packets were being handled by the TM, while the actual packet is buffered by the NPU processing engine. With the adopted model, a packet is forwarded to the egress port when its PDI is received by the NPU dispatch unit.

Algorithm 1 illustrates the overall algorithmic operation of the proposed TM. The TM operates in three phases: first, the policer checks if the received PDI is legitimate. The policer drops a packet if it is not valid (lines 1-2), or its flow is considered abusive according to its timestamp and the algorithm of Figure 5.4 (lines 3-5). Second, the scheduler tags each packet with its estimated schedule time (line 6), and asks to push it into the queue (lines 8-9). The

Algorithm 1: Flow-Based Traffic Management

Input: PDIin
Output: PDIout
// Phase one: FMT-policer
1: **if** (not PDIin.isValid)
2: Drop PDIin and skip to phase three;
3: **ref** flow ≡ FMT[PDIin.flowID];
4: **if** (decision is to drop, from QM_status and flow.Ts)
5: skip to phase three;
// Phase two: FMT-scheduler/shaper
6: Tag PDIin with flow.Ts and remove validity bit;
7: flow.Ts += PDIin.size × flow.bandwidth^{-1};
8: Set *push* to active;
9: Send the new PDI (PDIin) to queue manager;
// Phase three: queue manager
10: **if** (top PDI in QM is ready to exit or external dequeue activated)
11: Set *pop* to active;
12: Set QM_action to enqueue, dequeue, replace or no operation according to *push* and *pop* states;
13: Check QM_status for packet to be dropped if any;

shaper computes the schedule time for the next packet of the same flow (line 7). Finally, the queue manager will pop the top packet either if its scheduled time is reached or an external dequeue from the TM is activated (lines 10-11); the push requested by the scheduler and the pop requested by the queue manager are done synchronously (line 12). If the queue status is full with enqueue operation activated in the QM, the last packet in the queue is sent to the drop port (line 13). The traffic manager architecture with its surrounding modules are detailed in the subsequent subsections.

5.3.2.1 Flow Management Table

The TM includes a single FMT in which each record contains state and configuration parameters. The state is a timestamp (Ts) that is the expected time when the next packet of the flow can be sent to the egress port, and depends on the packet's flow. The configuration is the inverse of the allocated bandwidth (Alloc. BW), measured in bit-time, programmable according to required flow restrictions. Figure 5.3 depicts the interconnection of the FMT with the different TM modules.

A single FMT is sufficient, because the priority of packets is implicit to the characteristics of their flow and their bandwidth usage. The packets are simply ordered according to their

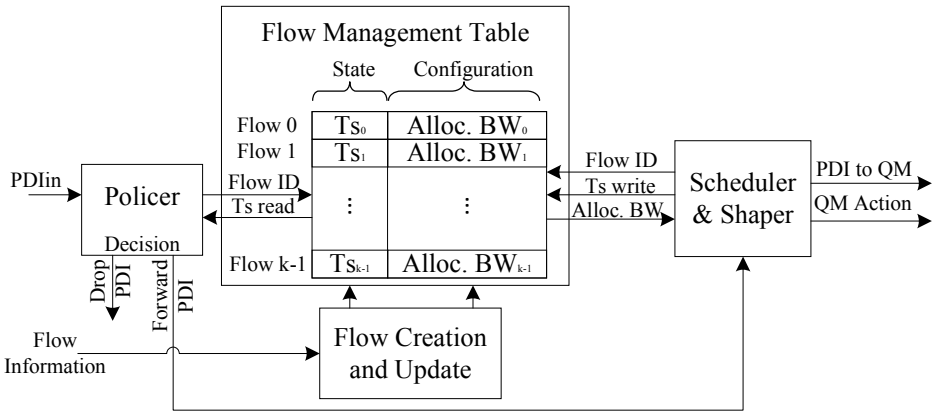


Figure 5.3 General block diagram interconnect of the traffic manager modules.

scheduled departure times. Note that, contrary to classic TM architectures [40, 60], this TM architecture avoids a hierarchy of queues by leveraging the flow number associated with each packet.

The proposed traffic manager functionalities are shown in Figure 5.4 that presents functionalities related to (a) policing, (b) scheduling, and (c) shaping. These functionalities are further detailed in the next subsections.

5.3.2.2 Policer

Policing ensures that traffic does not exceed certain bounds. In this work, the policer acts like an arbiter/marker with the following considered control actions, or namely the policing mechanisms:

(i) Drop a packet without enqueue, preventing congestion situations and overflow of the TM queue.

(ii) Drop a packet from the QM while enqueuing.

(iii) Forward incoming traffic if queue capacity allows it, i.e., there is room for the incoming packet.

The policing decision algorithm is a heuristic based on the packet timestamp record from the FMT and queue occupancy status, as depicted in Figure 5.4 with TM policing functionality (a).

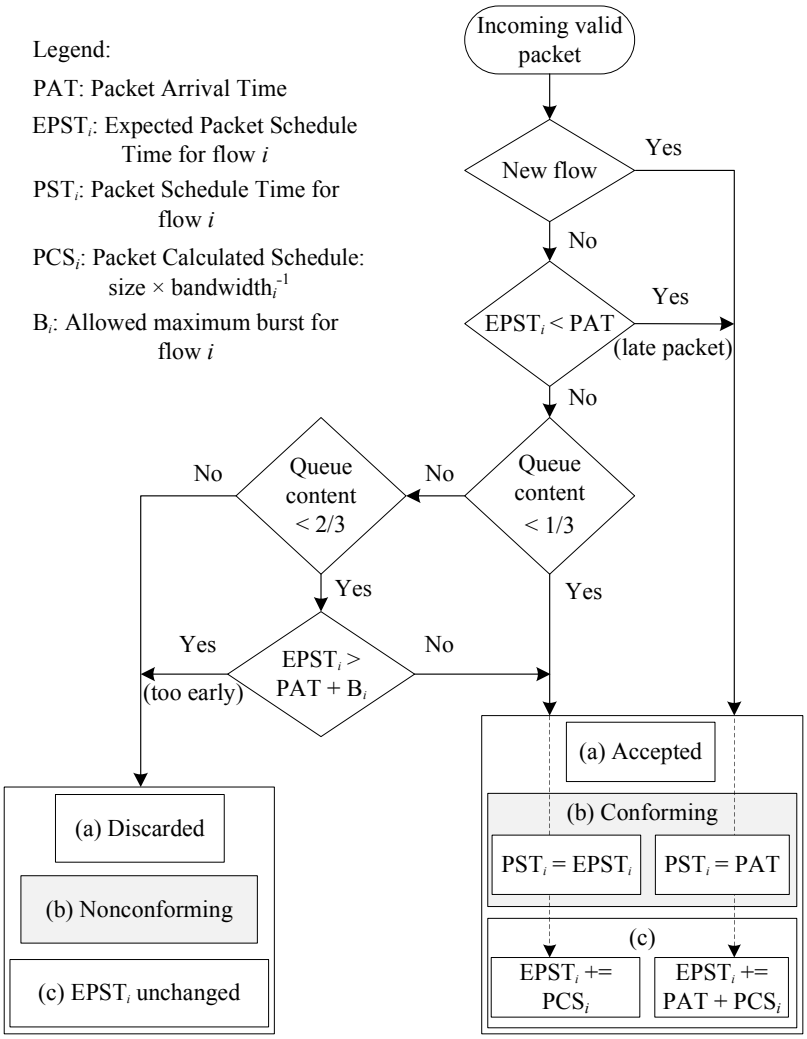


Figure 5.4 The proposed TM functionalities: (a) policing, (b) scheduling, and (c) shaping.

For the first mechanism (i), the policer acts before the packet is enqueued (so this control mechanism is an active congestion scheme). It checks if the packet is allowed based on the FMT records as stated in Algorithm 1 (lines 1-2), and to prevent queue overflow (lines 3-5). Once the packet passes the policer, it enters the next stage (scheduler/shaper).

The second mechanism (ii) is activated while the QM status is full, and the incoming packet PDI is legitimate. In that case, the last packet in the queue is dropped to have room to accommodate a legitimate packet as depicted in Algorithm 1 in line 13 (so this control mechanism is a passive congestion scheme). The third mechanism (iii) reflects the normal operation of compliant flows entering the TM. It is activated when the first mechanisms (i) and/or (ii) do not apply in a given cycle.

Policer mechanism (iii) enables to absorb bursts by checking the queue occupancy in real-time, if enough room exists, the burst is allowed to enter the TM until 1/3 of queue occupancy (first threshold) where packet are virtually green in analogy to three color markers [36]. Above this threshold, we start applying the burst limit for each specific flow, up to 2/3 of queue occupancy (second threshold, packets are virtually yellow). Beyond the second threshold, the first policing mechanism (i) is applied more aggressively. Only compliant flows are granted entry according to their expected arrival time (stored Ts records of each flow in the FMT, see Figure 5.4). Thus, packets are virtually red and discarded as nonconforming to prevent overuse of the bandwidth and network congestion.

5.3.2.3 Scheduler

The proposed TM scheduling functionality is depicted in Figure 5.4(b). The purpose of scheduling is to tag each PDI prior entry into the QM as depicted in Algorithm 1 (line 6). This schedule represents the earliest time at which the packet should be sent back to the network. Tagging the incoming packet plays an important role in avoiding that low priority packets be dequeued before the higher priority ones. Also, the same holds for older existing packets versus the current incoming ones. This time tag is calculated from the shaping policy (detailed in the next subsection).

5.3.2.4 Shaper

After the policer stage, each received packet is tagged with a timestamp. Packets timestamps of different flows are computed according to Figure 5.4 with the TM shaping functionality (c). The Ts value stored in the FMT depicts the earliest moment in time that the PDIin has to wait in the QM before it can be dequeued (PST), while the new computed Ts is the expected packet schedule time (EPST) for the upcoming packet for the same flow i. The calculated packet schedule time is in terms of clock cycles, and it depends on the size of the incoming packet and the corresponding inverse allocated bandwidth of flow i, as depicted in Algorithm 1 (line 7). This shaping enables the exact calculation of the EPST of incoming packets belonging to the same flow i, to follow and guarantee the requested flow bandwidth.

The adopted shaping policy enables a fair share of the bandwidth to the different flows, enforcing isolation through prioritization. For non-compliant flows trying to flood the network, their packets timestamps would be de-prioritized with this shaping policy, and therefore they would not affect the compliant one's as their Ts would be larger, i.e., they would have low priority. Nevertheless, once the upper threshold of the policer heuristic is reached (the allowed burst limit), with no room available to absorb this abusive flow in the QM (queue is 2/3 full),

policer's first and second control mechanisms (i, ii) are both activated. Packets belonging to the non-compliant flows will be dropped by the policer. They will be seen as part of a flow attacker or bandwidth abuser, and will not be allowed until the source's flow reduces its transmission rate, while complying with policer's first or second control mechanisms.

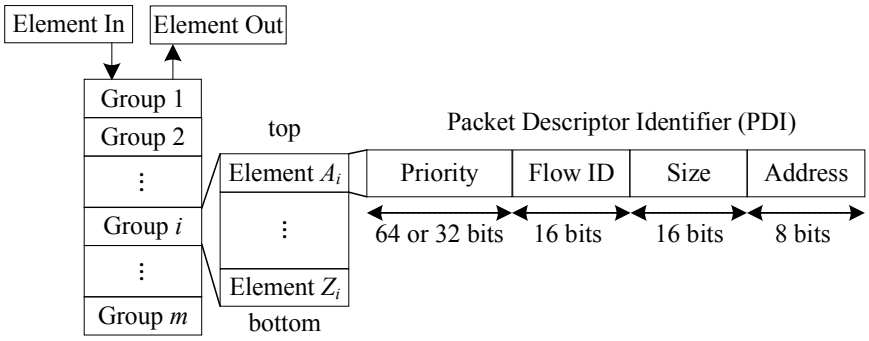


Figure 5.5 The hardware priority queue content.

5.3.2.5 Queue Manager

The most important component of the QM is the PQ that is responsible for enqueuing, dequeuing and replacing incoming packets in a single clock cycle. Also, the PQ sorts the packets in order of priority according to their timestamps in ascending departure order. More details about the queue management can be found in [9, 12].

Figure 5.5 depicts the PQ content. The PQ is divided in m groups. Each group contains N packets $A_g, \ldots Z_g$, where g is the group number. A_g and Z_g represent the min and max elements, respectively, of that group, and all remaining elements are placed in an unordered set $\mathcal{S}_g$. Namely, a group X_i contains N elements $\{A_i, \mathcal{S}_i, Z_i\}$ with $\mathcal{S}_i = \{X_i \backslash \{\min X_i, \max X_i\}\}$. The letters $A \ldots Z$ are used for generality, regardless of the actual number of packets, except in examples where N is known. Each group is being connected with its adjacent groups, each independently applying in parallel a common operation on its data. This PQ architecture is register-based single-instruction–multiple-data (SIMD), with only local data interconnects, and a short broadcasted instruction.

The priority queue accepts a new entry or returns the packet tag (PDI) with the lowest timestamp every clock cycle. The three basic operations supported by the priority queue are enqueue, dequeue, and replace (i.e., a combination of dequeue-enqueue), while the packet movement obeys Algorithm 2 for each defined queue operation representing packet selection and sort inside each distinct queue groups. From the performance analysis of this hardware PQ from [9], the performance decreases in $O(\log N)$, where N is the number of packets in

Algorithm 2: Hardware Priority Queue Operations

Input: PDIin (Element In)
Output: PDIout (Element Out = A_1, but not pertinent on enqueue)
for all groups i ($i = 1, 2, \ldots m$) **do**
// *On enqueue operation:*
group $i \leftarrow order\{Z_{i-1}, A_i, \mathcal{S}_i\}$;
// *On dequeue operation:*
group $i \leftarrow order\{\mathcal{S}_i, Z_i, A_{i+1}\}$;
// *On replace operation:*
group $i \leftarrow order\{\max\{Z_{i-1}, A_i\}, \mathcal{S}_i, \min\{Z_i, A_{i+1}\}\}$;

Where:

- *order* $X = \langle \min X, X \backslash \{\min X, \max X\}, \max X \rangle$
- $\mathcal{S} = \{X \backslash \{\min X, \max X\}\}$
- Z_0 is the incoming packet (Element In),
- A_{m+1} is the "none/invalid" packet equivalent to empty cell which must compare as greater ($>$) to any valid packet,
- $\max\{Z_0, A_1\} = Z_0$ during replace, since A_1 is dequeued,
- From invariants 1 and 2 [9], we have $A_1 \leq (\mathcal{S}_1) \leq A_2 \leq (\mathcal{S}_2) \leq A_3 \ldots$, etc. while the Z_i's are gradually ordered.

each group, while the quality of dismissed elements when the queue is full is $1/N$ (lower is better). In this work, N is fixed to 2 packets in each queue group, for all queue sizes. The PDI timestamp at the top of the queue (highest priority element) is compared to the current system time (in clock cycles). If the PDI Ts is reached, the *pop* signal is activated (Algorithm 1, lines 10-11), and the queued elements are re-ordered according to their schedule time (Algorithm 2). Also, an external pop can be issued, for example in case the packet dispatcher is idle. The packet at the top of the PQ is sent to the NPU packet dispatch unit to be dequeued from the packet buffer, and sent back to the network either to an egress port or the switch fabric interface (see Figure 5.2).

5.3.2.6 Programmable Traffic Manager

a) New flow creation

To allow creating new flows and erasing the record of inactive ones, the Ts in the FMT can be updated in real-time. This feature enables to create new flows in the FMT

and override any previous records without requiring FPGA re-synthesis. This is done by updating directly the on-chip Block RAMs (BRAMs) through the create/update port. This latter information is either forwarded from the control plane or the packet processor to the TM through the create/update flow information port. During this phase, the PDIin port should contain all necessary information for flow creation.

b) Flow bandwidth update

During operation of the TM, the bandwidth of one or a set of flows can be increased/decreased by the network operator, or as requested by the application requirements, a change in the QoS, or to exploit the unused bandwidth of the inactive flows. Updating the inverse allocated bandwidth records can be done simultaneously while processing the incoming traffic, with no performance impact. This is done through the use of dual port memories, enabling a single read and write in the same clock cycle to process and update different incoming flow bandwidth traffic information. During this phase, the create/update flow information port should contain all necessary information for flow update.

c) P4 support with extern modules

$P4_{16}$ supports integration of specialized hardware through `extern`. The TM is coded in C++ and can be easily ported into P4 program as an extern object/function, and attached to a flexible and programmable pipeline. The TM can be seen as an external accelerator attached through an extern control interface [30, 61].

In a programmable pipeline, P4 programs can request the operation implemented by the extern object/function (for example the TM) as depicted in Figure 5.6. The functionality of the TM is not specified in P4, but only the interface is. The interface of the extern object/function can be used to describe the operation it provides, as well as its parameters and return types. This interface is generally exposed to the data plane. It should be noted that the P4 program can store and manipulate data pertaining to each packet as user-defined metadata directly with the interface to the TM, without using the intrinsic metadata (control/signals) as defined in the P4 language specification [61].

5.3.3 General Packet Scheduling Schemes

Packet scheduling schemes can be categorized in two classes: timestamp-based that achieve good fairness and delay bounds, but that suffer from high computational complexity, and round-robin based that are simpler, but that suffer from large delay bounds. Further, another

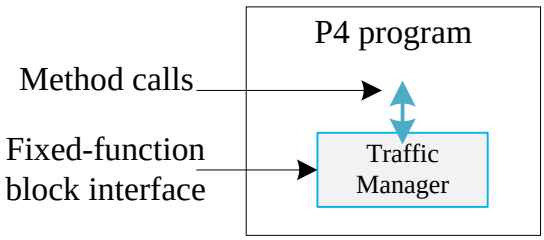


Figure 5.6 Block diagram representing a P4 program and its extern object/function interface.

classification is according to the work conserving nature of the scheduler, i.e., the link is never idle whenever a packet remains in the queue. On the other hand, a non-work conserving scheduler will not service a packet even though the link is idle due to a scheduling policy, or whenever the scheduling time is not yet met [36].

Our proposed scheduler is timestamp-based, non-work conserving, as packets will be served only when their schedule time is reached (dequeued from QM). To be able to service packets at idle link, an external dequeue from the TM should be issued if the link is idle, to service the top packet in the queue as detailed in Section 5.3.2.5.

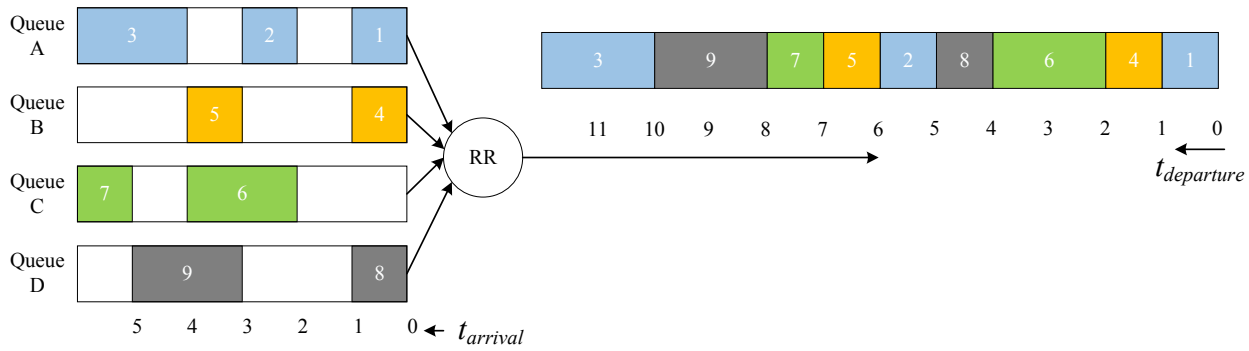
5.3.3.1 Examples of supported scheduling schemes

Our proposed scheduler/shaper can support different existing scheduling schemes like Round-Robin (RR) and Weighted Round-Robin (WRR), while supporting strict priority scheduling by default as the QM is built around a PQ.

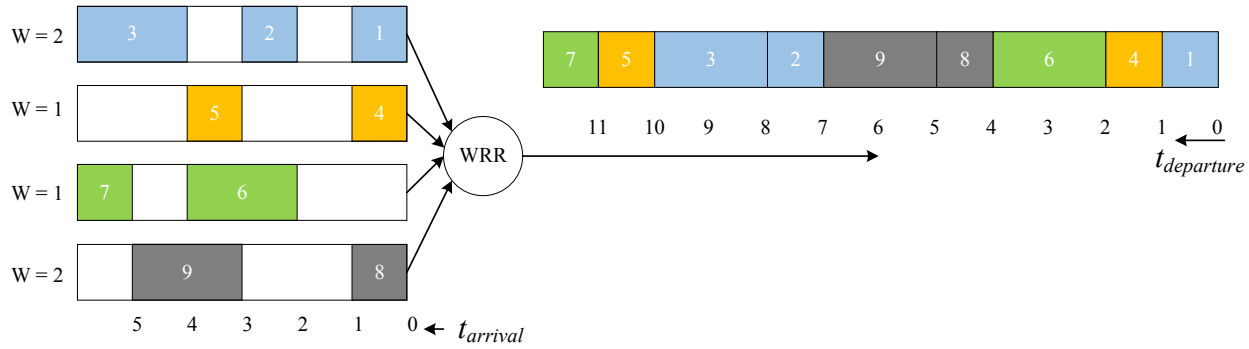
Let us consider the examples depicted in Figure 5.7a with RR, Figure 5.7b with WRR offering bulk service, and Figure 5.7c with WRR offering smooth service scheduling schemes.

In RR-based scheduling, each queue is served once during the scheduler iteration or round. This is one of the simplest example of scheduling to implement as timestamps of different packets received from each queue are incremented by the number of existing queues. In the example of Figure 5.7a, the four-queue system from top to bottom A, B, C, and D, queue A packets would have Ts as 0, 4, ... etc., 2nd queue B would have Ts as 1, 5, ... etc., and so forth for each received packet. Hence, to schedule packets according to RR, we should simply initialize the four first Ts flows of our scheduler to 0, 1, 2, 3 and increment the upcoming packets Ts's by the number of queues (4).

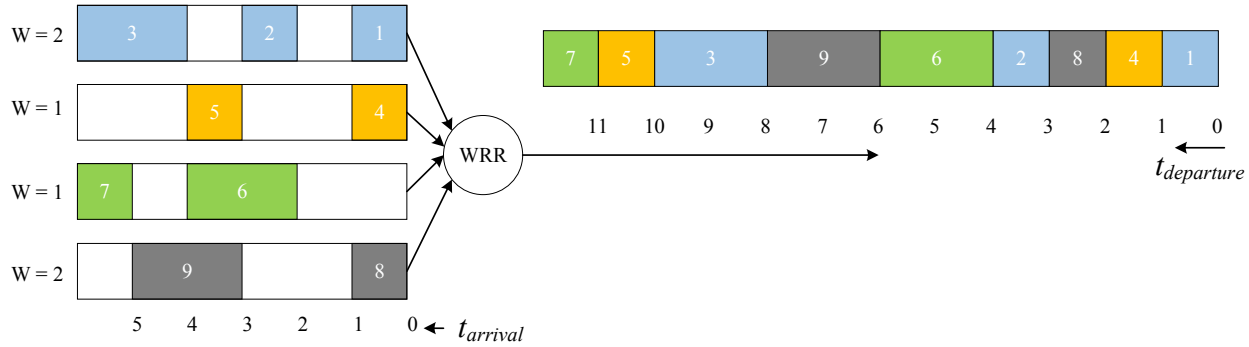
In the same way, WRR can be implemented through proper Ts initialization and weight adjustment, especially for the bulk service. For the example of Figure 5.7b, packets are



(a) Round-Robin.



(b) Weighted Round-Robin: bulk service.



(c) Weighted Round-Robin: smooth service.

Figure 5.7 RR and WRR scheduling schemes. (a) Round-Robin. (b) Weighted Round-Robin: bulk service. (c) Weighted Round-Robin: smooth service.

transmitted as AABCDD as queues A and D are each served two times over six (33%), while queues B and C are served one time over six (17%). So, we should initialize the four first Ts flows to 0, 2, 3, 4. For the first and fourth flows (queue A and D), the upcoming packets Ts's increment factor should alternate between 1 then 5, while packets of the second and third flows are incremented by a constant 6. With these increment factors, the scheduler

is able to offer the sequence AABCDD. It should be noted that supporting this alternation may require extra logic during implementation for the bulk service scheduler. For fairness reasons, smooth service is desirable with the ABDACD sequence. The latter sequence can be implemented through Ts's initialization to 0, 1, 4, and 2 while the increment factor is 3 for A and D, it remains 6 for B and C. For the above schedulers, the external dequeue is sufficient with no need to use the system time to issue a dequeue in the QM (see Section 5.3.2.5).

One of the most popular scheduling scheme is the weighted fair queuing (WFQ), as it approximates the ideal Generalized Processor Sharing (GPS) scheduler. However, due to its O(n) complexity, where n is the maximum number of nonempty queues, the required calculations at each packet arrival or departure in the WFQ are very expensive. Other packet fair queueing (PFQ) algorithms have been developed in order to reduce this complexity, as detailed in [14]. A good approximation to WFQ is Start-Time Fair Queueing (STFQ). The following shows how we can support this scheduler in a way that differs from [72]. The STFQ requires a virtual start time before a packet is enqueued. The virtual start time is computed as the maximum between the virtual finish time for flow i and the virtual time. The virtual finish time is the timestamp read from the FMT for the packet of flow i in our case, and the virtual time is the last dequeued packet Ts across all flows, that is the last dequeued Ts packet from the QM. So, STFQ requires only an additional comparator for selecting the maximum Ts. This comparator will not impact the performance during implementation as the required information is already available from the FMT and QM.

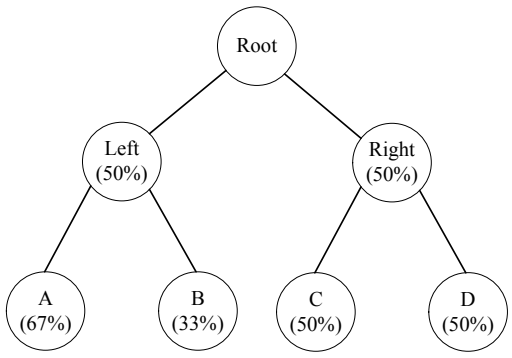


Figure 5.8 Hierarchical packet fair queueing.

For hierarchical scheduling, we can support hierarchical (H-PFQ) with rate limitation guaranteed, as our QM does not allow modification of already enqueued packet timestamps upon arrival of future packets. Let us consider the example depicted in Figure 5.8. If all queues A, B, C, D are nonempty, the service rate is 33% for A, 17 % for B and 25 % for C and D. If queue A is empty or inactive, B would be served 50 % of the time according to the left

node bandwidth. In Sivaraman work [72], hierarchical scheduling is done by using a tree of Push-In First-Out (PIFO) queues. However, in our case, with a single queue model, our assigned weights are programmable (through flows bandwidths). So, it only requires updating the flow bandwidth of queue B according to the traffic conditions, as a solution to use the unexploited bandwidth of queue A packets. On the other hand, when the packets of queue A arrive, they can start a bit earlier than the current time of B packets to re-balance the portions of bandwidth usage between A and B.

5.3.3.2 Discussion

Sivaraman [72] proposed a programmable packet scheduling architecture using Atoms processing units representing a programmable switch's instruction set. In Sivaraman work, the scheduling model is composed of two components: 1) a tree of PIFO queues. A PIFO is a PQ that allows elements to be enqueued into an arbitrary position based on the element's rank, but dequeues elements from the head. 2) the computation of an element's rank is done before it is enqueued into the PIFO, this computation is called a packet transaction [71].

In comparison to Sivaraman scheduler, our scheduler behave in the same way as to tag each received packet with a timestamp prior entry to the QM. However, in Sivaraman work, the rank computation is done through Atoms processing units running the scheduling code. In this work, we target an FPGA platform from which each flow has an initial timestamp and an inverse allocated bandwidth that can be updated at run time. More details about scheduling schemes support can be found in Section 3 of Sivaraman paper [72] as our schedulers are comparable. It should be noted that in this work we focus on traffic management and not specifically on scheduling.

5.4 Handling Timestamp Wrap Around

In general, the range of priority key values used in the PQ is much smaller or comparable to the actual range of stamps used in the scheduling module. Therefore, priority values may wrap around (overflow). For example, timestamps that handle packet transmission times with 1 GHz rate will wrap around every 4 seconds for 32-bit keys. Having both pre- and post-wrap around timestamps present in the PQ would result in order errors, i.e., post-wrap around timestamps will be regarded as smaller. We propose to mitigate this issue by considering the following solutions.

5.4.1 Use of a Wider Key

The configuration of the data type for the timestamp can be changed to a variable length type in the TM code through the arbitrary precision package from Xilinx. A wrap around every year would happen with 55-bit variable length, two years with 56 bits, etc., with the above clocking frequency (1 GHz). With 64-bit priority keys, the wrap around would happen every 585 years. It is a fair assumption that the circuit will not be operated without interruption for a period that long, and 64-bit might even look exaggerated, but this solution is taken as a reference to estimate maximum resource usage and lowest design performance. Wider keys require more storage capacity and wider comparators that eventually impact the critical path of the design.

5.4.2 Use of a 32-bit Key

The use of 32 bit key length has the advantage of halving the total length of the Ts in the PDI field. This impacts directly the memory usage in the FMT. Also, the complexity of the hardware used in the QM is reduced, especially the multiplexers and comparators. However, the use of 32-bit key requires adequate recalculation of the flow timestamp records in the FMT, before each wrap around.

This wrap around calculation is done as follows. Prior each system time wrap around, the FMT Ts records are recalculated when the system time exceeds the wrap around threshold (T_{wa}) according to (5.1), where B_{max} is the maximum burst threshold of all flows, and $\#\ FMT\_flows$ is the total number of flows supported by the FMT.

$$T_{wa} = (2^{32} - 1) - (B_{max} + \#\ FMT\_flows) \tag{5.1}$$

Each FMT Ts record exceeding the T_{wa} is recalculated by subtracting from it the current system time, while the other flows Ts's under the threshold have their records initialized to zero. In the present case, during FMT Ts records recalculation, the incoming PDI's are stored temporarily inside a buffer (the length of this buffer corresponds to the total number of supported flows in the FMT), to prevent order errors in the QM between already recalculated Ts and the other waiting flows, while the other packets in QM are served normally. The TM resumes its normal operation as soon as all records are recalculated. Assuming a FMT supporting 1024 flows, the TM would take 1024 cycles to recalculate the new timestamps prior resuming its normal operation after a time wrap around. If the TM is running at 100 MHz, a wrap around would happen every 42 seconds, and during the recalculation phase an incoming PDI has to wait in the temporary buffer 10 µs to be processed in the worst case

scenario. Also, the QM should be empty to resume the TM normal operation.

5.5 HLS Design Methodology and Considerations

In this section, we first present the analysis of operations required by the proposed TM design. Then, we detail the steps we apply in HLS to obtain the desired throughput and latency.

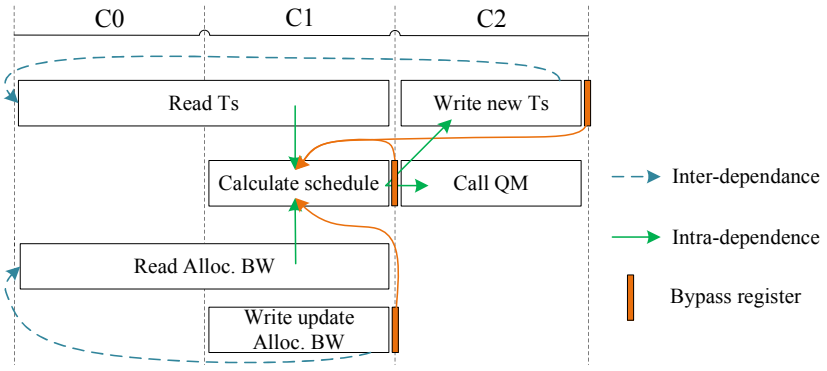


Figure 5.9 Proposed TM pipeline operations timing diagram.

5.5.1 Analysis of Traffic Manager Operations

The timing diagram demonstrating correct operation of the proposed TM is shown in Figure 5.9. The required operations for the TM to process any incoming PDI (representing concise packet information) are to check the FMT record for the specific incoming flow Ts and queue occupancy status, make a decision to drop or forward it, update the FMT flow's record, and finally send it to the QM with a Ts tag. Therefore, the TM operations consist in reading Ts memory (steps C0-C1), calculating the new schedule time (step C1), and writing it back to the same memory location (step C2). Moreover, a FMT bandwidth (Alloc. BW) access is required with a read and/or write (update) during steps C0-C1. Finally, the PDI is forwarded to the QM according to policer's decision in step C2. These are the specific tasks done by the proposed TM for each incoming PDI at any given clock cycle.

5.5.2 Design Methodology

The HLS process is initiated by specifying the C++ design files, a target FPGA device, and appropriate directives and constraints to apply to the design (that are specific to the HLS

tool). The HLS process can be described in three steps:

1. Extraction of data and control paths from the high-level design files.
2. Scheduling and binding of the RTL in the hardware, targeting a specific device library.
3. During the above step, optimizations are dictated by the designer to guide the HLS process, through specific directives and constraints.

From Figure 5.9, it can be seen that the minimum latency that can be achieved from our design operation is two cycles, with an initiation interval (II) of 1 clock cycle, i.e., every clock cycle an output PDI is ready. Thus, to target this optimal performance through HLS, the three directives that we focused on are: 1) a latency directive targeting 2 clock cycles, 2) a pipeline directive targeting an II of 1 cycle, and 3) a memory dependency directive asking for two separate true dual port memories for accessing the Ts and flow bandwidth records in the FMT. As HLS constraint, we target the lowest feasible clock period without violating the desired design latency and II mentioned above. It should be noted that an adequate access memory bypass is required if back-to-back similar PDIs of the same flow are received cycle after cycle, at minimum initiation interval. The reason is that Alloc. BW and Ts are required in the first stage, while they are updated in the second and third stages of previous PDI (see inter-dependences in Figure 5.9). Hence, we designed a two-stage bypass for the Ts memory and one-stage bypass for the flow bandwidth memory, respectively. The achieved implementation results are detailed in Section 5.6.

5.6 Implementation Results

In this section, we detail the hardware implementation of our proposed TM architecture, resource usage and achieved performance, for different configurations (64-bit and 32-bit priority key with 40-bit metadata). Then, comparisons to existing works in the literature are discussed. Finally, the hardware validation environment is presented.

5.6.1 Placement and Routing Results

The proposed TM was implemented on a Xilinx ZC706 board (based on the xc7z045ffg900-2 FPGA), and a complete design was also produced for a XCVU440 Virtex UltraScale device (xcvu440-flgb2377-3-e FPGA). Resource utilization of the entire TM architecture for different QM capacities was characterized in terms of the number of supported PDIs and the obtained performances. Results are shown in Table 5.2 for designs with 64 and 32-bit priorities, $N = 2$

Table 5.2 Resource utilization and achieved performance of the proposed traffic manager with 64 and 32 priority key bits on different platforms

Platform	Resources & Performances	Traffic manager architectures: 64-bit (no wrap around) Queue capacity (number of PDIs) 64	128	256	512	32-bit (wrap around) Queue capacity (number of PDIs) 64	128	256	512
ZC706 Zynq-7000	BRAM	5 / 1%	5 / 1%	5 / 1%	5 / 1%	3 / 1%	3 / 1%	3 / 1%	3 / 1%
	DSP	3 / 1%	2 / 1%	2 / 1%	2 / 1%	3 / 1%	3 / 1%	3 / 1%	3 / 1%
	LUTs	13465 / 6%	26960 / 12%	50145 / 23%	99043 / 45%	9218 / 4%	16650 / 8%	32719 / 15%	65446 / 30%
	FFs	7209 / 2%	13895 / 3%	27207 / 6%	53830 / 12%	5078 / 1%	9684 / 2%	18900 / 4%	37334 / 9%
	Frequency (MHz)	119.5	119.2	119.7	119.3	121.5	122.7	119.7	119.3
	Latency (cycles)	2	2	2	2	2	2	2	2
	Initiation Interval (cycles)	1	1	1	1	1	1	1	1
	Dynamic Power (W)	0.30	1.11	1.08	1.84	0.20	0.36	0.62	1.12
XCVU440 Virtex UltraScale	BRAM	5 / 1%	5 / 1%	5 / 1%	5 / 1%	3 / 1%	3 / 1%	3 / 1%	3 / 1%
	DSP	2 / 1%	2 / 1%	2 / 1%	2 / 1%	2 / 1%	2 / 1%	2 / 1%	2 / 1%
	LUTs	13321 / 1%	26448 / 1%	51811 / 2%	97338 / 4%	9085 / 1%	16736 / 1%	33814 / 1%	71121 / 3%
	FFs	7207 / 1%	13863 / 1%	27176 / 1%	53798 / 1%	5077 / 1%	9684 / 1%	18900 / 1%	37332 / 1%
	Frequency (MHz)	153.1	150.5	150.7	150.4	154.8	164.6	155.6	152.1
	Latency (cycles)	2	2	2	2	2	2	2	2
	Initiation Interval (cycles)	1	1	1	1	1	1	1	1
	Dynamic Power (W)	0.81	1.31	2.44	4.16	0.45	0.75	1.40	2.77

(the number of PDIs in each group of the queue), and a FMT supporting up to 1024 distinct concurrent active flows. It should be noted that we can support up to 1024 flows in all implementations. Supporting up to 1024 flows is a design decision and is not imposed by a limitation of FPGA BRAM resources. This number of flows was deemed sufficient based on the analysis reported in [86], while flows are identified from the 5-tuple information. More flows can be supported if that parameter is suitably set prior to the design HLS, placement and routing.

In the reported TM implementation, only flip-flops (FFs) and look-up tables (LUTs) were used in the QM module to obtain a fast and pipelined architecture. On-chip memory (Block RAM_18K) is used only in the FMT module. The achieved clock is less than 8.40 ns on the ZC706 platform, for 512 deep queue capacity, with both 64-bit and 32-bit priority TM architectures. When targeting the XCVU440 FPGA, the achieved clock is less than 6.72 ns for both TM architectures, with the latter queue capacity. The 32 bit architecture consumes 34% fewer LUTs and 31% fewer FFs than the 64-bit TM architecture, for both the ZC706 platform and XCVU440 FPGA device.

The achieved initiation interval (II) is one PDI per cycle, while the TM throughput is 80 Gb/s for both 64 and 32-bit architectures under the ZC706, for 84 bytes minimum size Ethernet packets (including minimum size packet of 64 bytes, preamble and interpacket gap of 20 bytes). Under the XCVU440 UltraScale FPGA, the achieved TM throughput is 100 Gb/s for both 64 and 32-bit architectures. The design latency is 2 clock cycles, i.e., the TM is fully pipelined and each incoming PDI takes a constant 2-cycle to be processed. It should be noted that accessing a memory location and updating it in the FMT takes at least 2 clock

cycles (as explained in Section 5.5.2 and Figure 5.9) on the target FPGA. This constraint is critical as the core operation consists of a read or read-modify, followed by a write to a memory. While writing the result to the memory, the QM is activated to reduce the design latency, explaining the necessity for 2 clock cycles, which is achieved by HLS with minimum design efforts and more flexibility, enabling faster design space exploration than hand-written RTL designs.

The total dynamic power consumption when targeting the ZC706 is estimated by the Vivado tool at 1.84 and 1.12 W respectively for the 64 and 32-bit architectures, which represents a 39% reduction for the latter. When targeting the XCVU440 UltraScale device, the power usage is reduced by 33% between the TM architectures (see Table 5.2) for a 512 queue capacity. The power consumption is dominated by the QM array. For example, the 512×104 or 512×72 queue bits for 64 and 32-bit architectures represent 90% of the dynamic power usage, when minimum packets sizes of 64 bytes are received at each cycle (back-to-back). Let us recall that the QM contains a PQ that is a highly parallel regular array of registers and comparators. The total queue capacity that can be supported by the XCVU440 FPGA is around 13.2k PDIs with 64-bit and 18.1k PDIs with 32-bit priority keys. From Table 2.2, comparing the reported results with other traffic management solutions under different platforms, the achieved TM performance can be compared to those obtained with design expressed at lower level hardware description languages (HDLs) [2, 3, 21, 32, 40, 42, 64, 79, 86].

Table 5.3 Memory, speed and throughput comparison with queue management systems

Queue Management System	Memory hierarchy	Resource Utilization			Performance			Platform
		FFs	LUTs	BRAM	Speed (MHz)	Latency (cycles)	Throughput (Gb/s)	
OD-QM [86]	On-chip	828	2505	56 × 36kbit	133	9	8	Virtex-5 XC5VSX50t
QMRD [44]	On-chip	5562	5798	389 × 18kbit	—	—	< 9	Virtex-II QDR
NPMADE [54]	External-SRAM	4370	6755	17 × 18kbit	125	10	6.2	Virtex II Pro
Proposed QM	—	53830	99043	—	119	1	61	ZC706
		53798	97338		150		77	XCVU440 UltraScale

Table 5.3 summarizes results obtained with various queue management architectures, knowing that the throughput of the QMRD [44] system depends on the protocol data unit (PDU) payload size, the reported OD-QM [60] results are for 512 active queues, and 64 bytes per packet. To make sure that our design is comparable, it was implemented with a total of 512 PDIs queue capacity, 64/32 bit priority, and the worst case egress port throughput is reported assuming 64-byte packets, supporting pipelined enqueue, dequeue and replace operations in a single clock cycle, i.e., O(1).

Compared to existing NPU solutions like Broadcom [21], and Mellanox NPS-400 [32], that can support up to 200 and 400 Gb/s respectively with built-in queue management systems, our

proposed TM architecture is scalable in terms of performance for different queue capacities. Using the single FPGA on a ZC706, we can support two 40 Gb/s links assuming minimum 64 byte sized packets, while with XCVU440 UltraScale, we can support four 100 Gb/s links with a QM that could reach 4.5k PDIs capacity per link. To scale up to 400 Gb/s with ZC706 boards, we can use several FPGAs in parallel like in a multicard "pizza box" system. Moreover, it should be noted that an FPGA is much more flexible than a fixed and rigid ASIC chip.

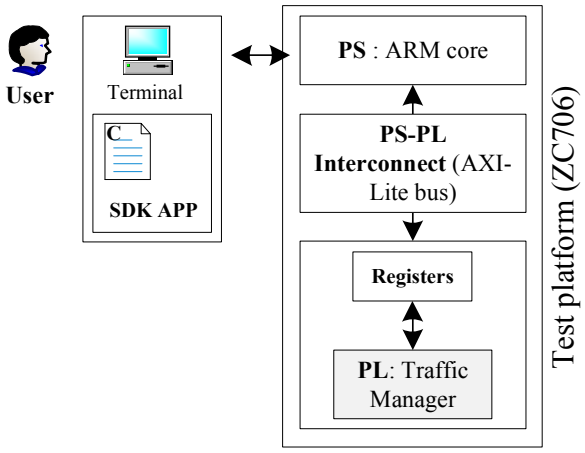


Figure 5.10 The TM hardware validation environment.

5.6.2 Hardware Validation Environment

To verify the correct functionality of the proposed TM after FPGA placement and routing, we tested the proposed 64-bit architecture on a Zynq-7000 ZC706 FPGA board from Xilinx. Figure 5.10 depicts the organization of the testing platform, it consists of four parts: the host computer user interface, known as the Xilinx Software Development Kit (SDK) tool, the processing system (PS-side), the PS-PL interconnect (AXI-bus), and the programmable logic (PL-side).

5.6.2.1 Host Computer User Interface

The user interface manages the data to display on-screen. This data is normally requested by a user from the PL-side. It represents the outputs (valid and dropped PDIs) from the TM at a given cycle. This data is read from the storage buffer on the FPGA board through a C program that runs on the PS-side. It is transferred to the processing system through a

built-in UART and handled by the SDK.

5.6.2.2 Processing System

The ZC706 board integrates a dual-core ARM. The ARM processor is clocked at 667 MHz and runs a native operating system. The main objective of the PS-side is to ease the process of data exchange between the user and the PL-side through the PS-PL interconnect, i.e., by the built-in AXI-bus. The ARM is used to manage data transfers from the FPGA part to the user interface.

5.6.2.3 PS-PL Interconnect

Communication between the PS-side and the PL-side is done using AXI-bus interface. For testing, we used the AXI-Lite bus interface known to offer low-throughput and low-latency communication capabilities [81]. It allows transferring the generated data obtained from the TM to the PS side. A slow bus solution is sufficient in this case as data is requested from the output buffers with one PDI each time.

5.6.2.4 Programmable Logic

We use the available logic resources in the FPGA chip to implement the TM architecture. The TM can accept a PDI every clock cycle, and can produce an output PDI in the same cycle. Transmitting such data flow to the host computer is impractical. One solution is to store the output PDIs from the TM in an output buffer, and then request them one by one later. The generated PDIs are known a priori, and they are displayed through the user interface. This allows analyzing specific characteristics of the design under test. For example, reports that could be generated can relate to back-to-back ingress burst handling, flow bandwidth abuse and policer dropping capability, over-exceeding the ingress port limit TM behavior, etc. These tests confirmed the correct functionality of the proposed TM and matched the co-simulation results that were detailed in Section IV of [8].

5.6.3 Future Work and Research Directions

Statistics gathering is one of the complex operation to perform as it lays in the critical path of packet processing of any network device. During development of the proposed traffic manager, statistics gathering for different flows was designed in a separate module external to the traffic manager with no interaction. This was chosen to avoid degrading the performance of the traffic manager, and have flow statistics reported once per second. Statistics gathering

could be integrated in the traffic manager. Alternatively, we could use dedicated metadata field for reporting flow state per received packet. This enables to have information about network flows in a cycle accurate manner, i.e., it can allow in-band network telemetry [50].

Another future research direction is to integrate the classification stage within traffic management. This could lead to faster creation/update of flow information from the control plane, classification and traffic manager stages. Also, it could facilitate the control, management and synchronization between different network equipment modules.

A recent trend in the literature led by the P4 language consortium is to integrate the traffic management in the programmable data plane [19]. In today's P4 programmable switches, traffic management is not supported directly in the data plane. An effort and thrust toward programmable traffic manager functionalities in today's network data plane is a near future target. This would be interesting to have a complete view of the system from classification, traffic management and packet buffering that are all programmable while user custom in-line processing would be supported by the P4 language directly in the network data plane.

5.7 Conclusion

In this work, we proposed, implemented and evaluated a high-speed, low-latency, programmable and scalable traffic manager architecture intended for flow-based networking. It is capable of providing all the functionality of typical network traffic managers from policing, scheduling, shaping and queuing. The proposed traffic manager architecture is coded in C++ providing more flexibility, and easier implementation than the reported works in the literature that were coded in VHDL, Verilog, etc. It is of interest to mention that the queue manager supports 64 or 32 bit priority keys with 40-bit of metadata representing the size, flow ID, and packet address, while the concise packet information tag is up to 104-bit.

The proposed traffic manager architecture was prototyped in FPGA using HLS and implemented with Vivado from Xilinx, targeting the ZC706 board and XCVU440 UltraScale device. The resulting design is capable of handling high speed network and links operating up to 100 Gb/s with minimum size Ethernet packets. Also, the flexibility of the architecture and the adopted high-level coding style facilitate introducing modifications and enhancements. For example, adding a congestion control mechanism like weighted random early detection (WRED) or using different types of queue in the queue manager, like binary heap, would be straightforward.

CHAPTER 6 GENERAL DISCUSSION

This thesis introduced new hardware architectures for network data plane queuing and traffic management for high-speed networking devices in the context of SDN. In this chapter, we discuss the proposed implementations and highlight the limitations of our work.

Initially, we proposed a SIMD hardware PQ (chapter 3). This PQ was selected for its guaranteed performance with constant latency and throughput. It supports the three basic operations: enqueue, dequeue and replace, that are executed in a single clock cycle. Our novel approach proposed to modify the sorting operation in a way to restore the defined queue invariants after each operation. This PQ architecture was coded in C++, and implemented in the ZC706 FPGA board. The results showed the scalability of the solution for different queue depths, and matched the theoretical expected results with the FPGA implementation for different performance and area metrics. Also, we managed to achieve using HLS best performances that are similar to those obtained with handwritten RTL designs, while supporting 100 Gb/s throughput with minimum 64-byte sized packets.

The main limitations of the SIMD PQ architecture are in the resource consumption, especially for the LUTs/FFs that are directly proportional to the queue depth or the number of groups. Also, the limited logic resources in an FPGA typically prevent exceeding a 1 Ki capacity when targeting the ZC706 board for a group size of $N = 64$. In addition, the quality of dropped elements when the queue is full is $1/N$ (lower is better), and the throughput decreases in $O(\log N)$. This SIMD PQ can be used for different tasks and applications, such as scheduling, and real-time sorting, etc. However, this architecture with partial sort cannot guarantee the order of departure according to the order of arrival of elements with similar priority.

To alleviate the limited logic resources in FPGA platforms, we proposed the HPQS (chapter 4) to increase the total capacity with guaranteed performance. Rising the capacity of the design is done by using available on-chip FPGA memories (BRAMs). The HPQS supports two configurations. The first one is for scheduling packets in a multi-queues system that can guarantee the order of departure according to the order of arrival, for elements with similar priority, using full sort. The second one is a high-capacity priority queue. The HPQS was FPGA-prototyped, while coded in C++. We achieved similar performances as those obtained with handwritten designs with pipelined queue operations, and constant one clock cycle latency. A thorough design space exploration was performed using the ZC706 FPGA board and XCVU440 Virtex UltraScale device for performance (throughput, latency, and clock period), area (LUTs, FFs, and BRAMs), and power consumption analysis for

different hardware PQ depths, HPQS heights and configurations. This analysis confirmed the scalability of the proposed solution, while we can support up to 512× the original largest SIMD hardware PQ in a single FPGA.

Further enhancements can be introduced in the HPQS, especially area utilization of the queue line elements counters can be improved (see Section 4.4). This can be done through the use of full and empty flags of 1-bit each, instead of 11-bit sized registers per queue line. This would allow reducing the complexity of the priority encoder for the push/pop indexes calculation. Also, we can reduce further the sorting information key in the HPQS type-1 configuration by 1-bit. Recall that the sort information in the hardware PQ was represented over $P - Q + 1$ bits (see Section 4.4.1). This 1 bit addition is used to differentiate valid sort information from invalid keys, i.e., empty elements. Mapping the sort information over $P - Q$ bits could allow reducing the routing from the storage area (BRAMs) to the hardware PQ and vice versa.

It should be mentioned that with the largest designs, the FPGA on the ZC706 board was filled beyond 80% in terms of LUTs, and BRAMs. This caused long synthesis and implementation times. Generally, this arises when routing resources are heavily used in the entire FPGA.

Different network traffic have distinct characteristics such as the flow rate, size, burstiness, etc. and requirements to meet as the QoS. Traffic management is used to achieve fairness and enforce isolation, prioritizing the different traffic flows while preventing packet congestion that can cause severe network problems. In network traffic management, we started first by developing the core functionalities of policing, scheduling, shaping while integrating the SIMD hardware PQ for sorting and storing purposes, that is crucial to keep this traffic scheduling at gigabit link rates. Moreover, the HPQS type-2 high capacity PQ can be used for the latter purposes. In our first complete TM design [7], a data dependency constraint prevented to fully pipeline the design achieving an initiation interval (II) of three clock cycles. This data dependency was between the load/store operations in the TM FMT. After analysis of the TM operations, and the introduction of proper directives in the design especially to allow the on-chip FPGA memories (BRAMs for FMT) read and write operations in the same cycle with no data dependency, the HLS tool achieved the target performances with an II of one clock cycle and minimum latency of two clock cycles [8]. While adding new features to the TM, we followed the HLS design flow optimization methodology depicted in Figure 2.6. Some of these features are: policer decision based on flow heuristics and queue occupancy status, flow creation and update in the FMT. HLS enabled us to achieve target performances similar to handwritten HDL designs.

The proposed TM (chapter 5) is capable of supporting pipelined operations, back-to-back packets of similar flows, processing minimum sized 84-byte Ethernet packets while providing 100 Gb/s throughput without loosing performance during flow updates (tuning). Also, in this TM, a single queue is used to order the packets according to their scheduled departure times. The priority of packets, i.e., early departure time, is related directly to their size and corresponding flow allocated bandwidth. Contrary to classic TM architectures with multilevel hierarchical queuing systems [3, 40, 60, 86], this TM architecture avoids a hierarchy of queues by leveraging the flow number associated with each packet. This is a benefit of flow-based networking.

The proposed TM can support today's programmable data planes through P4, as it can be integrated as an `extern` object/function. The TM can be seen as an external accelerator or coprocessor. As in today's P4 programmable switches, traffic management is not supported directly in the data plane. An effort and thrust toward programmable traffic manager functionalities in today's network data plane led by the P4 language consortium is a near future target [19].

Further enhancements in the traffic manager can be done in the wrap around flows timestamps recalculation (the system and schedule times are in clock cycles, measured in nanoseconds), without the need to buffer incoming packet tags and to stop normal TM operation during this step (see Section 5.4.2). This can be achieved by allowing packet tags to be stored in the QM for at most 2.14 seconds ahead of their expected schedule time (as a maximum burst limit B_{max}). Moreover, the stored packet schedule time should not exceed the maximum expressed over 32 bits inside the queue (i.e., 4.29 seconds). Once the top element in the QM reaches 2.14 seconds, the most significant bit over 32 bit key is set. This sets a need to reset it in the queue for all stored valid elements. Hence, all timestamps in the QM are recalculated. The FMT timestamps must be represented with more than 32 bits to allow proper substraction of each 2.14 seconds interval as the system time progresses before entry to the QM, this can be done as a XOR operation between 1 and the 32nd bit of the packet schedule time coming from the FMT-scheduler/shaper. Hence, the wrap around is done directly without any need to do FMT timestmps recalculation.

With my collegues in the LASNEP laboratoy (B. Fradj, T. Luinaud, J. S. da Silva, and T. Stimpfling), we developed a test platform based on the ZC706 FPGA baord supporting plug-and-play networking modules such as a parser, a deparser, a classifier, etc. as depicted in Figure 6.1. This test platform consists of 4 parts: (a) host computer user interface used to display the results received from the platform, (b) processing system (slow path) that handles sending control requests to the programmable logic and transmit the received results to the

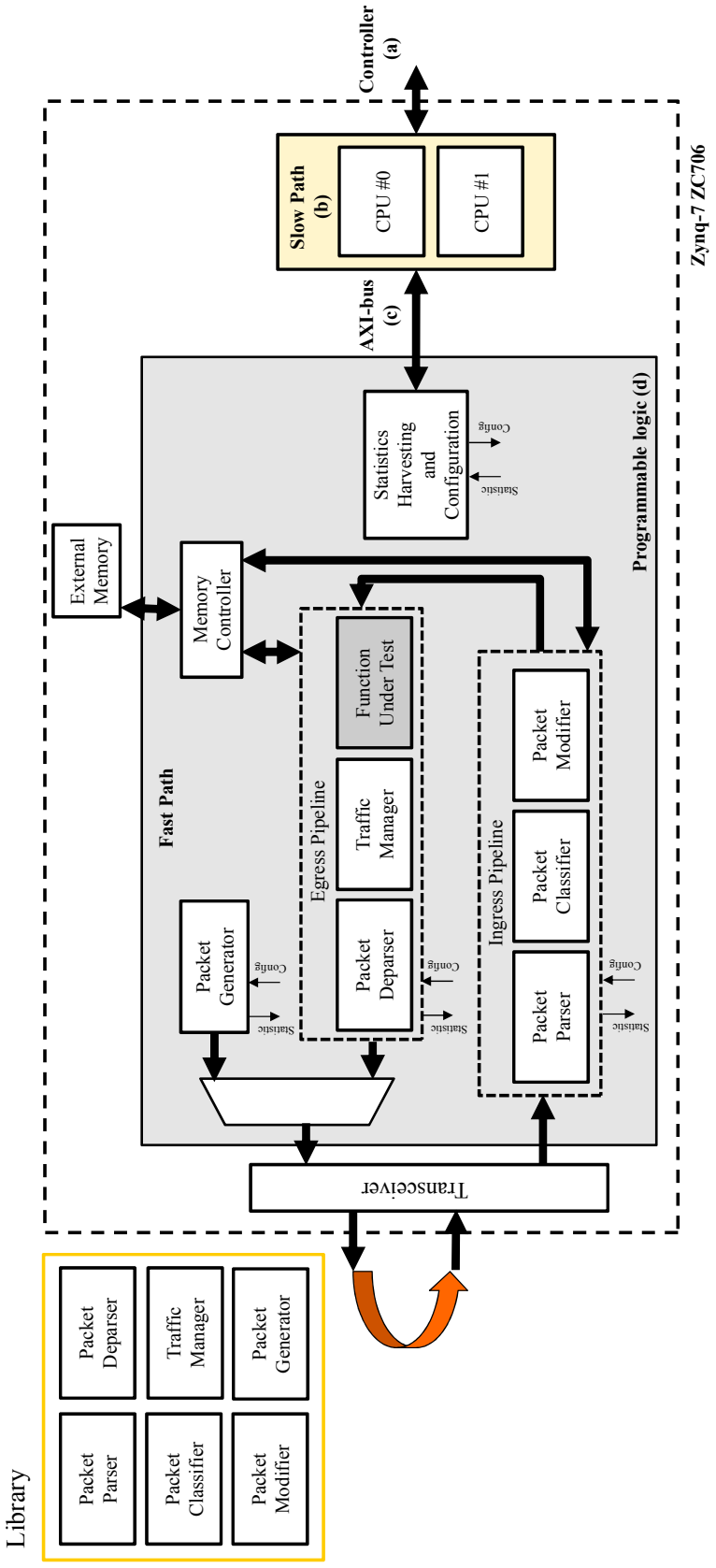


Figure 6.1 Network data plane test platform.

host computer, (c) the PS-PL interconnect (AXI-bus interface), and (d) the programmable logic (fast path) that can support networking functions in the ingress and egress packet processing pipelines. Our proposed traffic manager was integrated into this test platform to provide the necessary functionalities for traffic management, also to verify correct operation of the developed networking functions that can be selected from a library, or any newly developed networking function (function under test). After the configuration of the different modules from the slow path, the pseudo random packet generator sends packets to the the transeiver. The generated packets are received from the transceiver through an external loopback. The parser extracts the header packet information that are used by the classifier in the ingress pipeline, from which the packets are given specific flow numbers before entering the traffic manager. A decision is given by the traffic manager to either transmit or drop these packets to enforce QoS. If a packet is forwarded from the traffic manager, it enters finally a deparser before retransmission to the network. During operation of the different networking functions, packet statistics are collected from the ingress pipeline, egress pipeline, especially from the traffic manager such as the ingress, egress and dropped packet rates, in addition to the different allocated bandwidths for each flow and their transmission rates to see if being enforced. This test platform validated the correct functionality of the different developed networking functions.

CHAPTER 7 CONCLUSION AND FUTURE WORK

7.1 Summary of the Work

High-speed low-latency designs and implementations are the utmost requirements in today's networking devices. Moreover, two competing design methods can be used to achieve these requirements: High-Level Synthesis (HLS) versus careful design with register transfer logic–hardware description language (RTL–HDL). The former one targets minimum design efforts, more flexibility and faster design space exploration than the latter. HLS was chosen in this research due to the above reasons. Two distinct pure implementations can be derived from the same HLS design. The first one is expressed at high-level in C/C++ targeting a GPP (general purpose processor software solution). The second one is obtained by compiling the high-level description as a lower-level one expressed in VHDL/Verilog targeting an embedded system like an FPGA platform (a hardware solution).

The work described in this thesis makes several contributions. For instance, we proposed, designed, implemented and validated new architectures and accelerators for network data plane queuing and traffic management. We developed High-Level Design (HLD) descriptions of accelerators that are implemented through HLS. The HLDs are optimized and refined to achieve targeted low latency and high throughput. These designs focused on priority queuing, high capacity HPQS, and finally flow-based traffic manager implementations on FPGA targeting performance of low-level handwritten designs.

For network queuing, we proposed a SIMD hardware PQ to sort out packets in real time, supporting independently the three basic operations of enqueuing, dequeuing, and replacing in a single clock cycle, aiming for high-throughput and low-latency solutions. The implemented PQ architecture is coded in C++. Vivado HLS was used to generate synthesizable RTL from the C++ model. This implementation on the ZC706 FPGA board shows the scalability of the proposed architecture for various queue depths with guaranteed performance. The hardware PQ offers a 10× throughput improvement when compared to prior works.

To increase the queuing capacity, we presented the HPQS intended for scheduling and prioritizing packets in the network data plane. Due to increasing traffic and tight requirements of high-speed networking devices, a high capacity PQ, with constant latency and guaranteed performance is needed. The proposed HPQS enables pipelined queue operations with one clock cycle latency. Two configurations were proposed. The first one is intended for scheduling with a multi-queuing system for which we report implementation results for 64 up to 512

independent queues. The second configuration is intended for large capacity PQs that can reach up to ½ million packet tags using a single FPGA.

For traffic management, we proposed a flow-based TM. Flow-based networking allows treating traffic in terms of flows rather than as a simple aggregation of individual packets, which simplifies scheduling and bandwidth allocation for each flow. We presented a programmable and scalable TM architecture, targeting requirements of high-speed networking devices, especially in the SDN context. This TM is intended to ease deployability of new architectures through FPGA platforms, and to make the data plane programmable and scalable. It supports also integration to today's programmable data plane with P4, as a C++ `extern` object/function. This TM is capable of supporting links operating at 100 Gb/s while scheduling packet departures in a constant 2-cycle per packet.

7.2 Future Works

Even though the work described in this thesis has presented multiple contributions in network data plane queuing and traffic management. There is always room for improvements to the solutions presented, and many extensions could be provided.

During each queue operation, the elements in all groups of the SIMD hardware PQ are all being sorted. To decrease the power consumption that was found to constitute 90% of the dynamic power usage in the TM, empty groups should be deactivated. This can be done either by using the queue status in the TM or by checking the last element of any queue group (from invariant 3). If the last element is empty, then the next queue groups are all empty. Hence, the content of these groups remains unchanged and they could be deactivated to reduce power dissipation.

Statistics gathering is one of the complex operations to perform in traffic management as it lays in the critical path of packet processing. During development of the TM, statistics gathering for different flows was designed in a separate module external to the TM to reduce complexity. This was chosen to avoid degrading the performance of the TM, and have flow statistics reported once per second. Statistics gathering could be integrated internally. Alternatively, we could use dedicated metadata fields for reporting the flow state for each received packet. This enables to have information about network flows in a cycle accurate manner, that can allow in-band network telemetry.

Another future research direction is to integrate the packet classification stage of a NPU within its traffic management module. This could lead to faster creation/update of flow information from the control plane, classification and TM stages. Also, it could facilitate the

control, management and synchronization between different network equipment modules. Moreover, in today's networks, it is important to have an entire programmable data plane specified and managed using P4. This would be interesting to have a complete view of the system from classification, traffic management and packet buffering composed of modules that are programmable while user custom in-line processing would be supported by the P4 language directly in the network data plane.

REFERENCES

[1] Y. Afek, A. Bremler-Barr, and L. Schiff, "Recursive design of hardware priority queues," *Computer Networks*, vol. 66, pp. 52–67, 2014.

[2] Agere, "10G Network Processor Chip Set (APP750NP and APP750TM)," *Product Brief*, 2002.

[3] B. Alleyne, "Chesapeake: A 50Gbps network processor and traffic manager," in *IEEE Hot Chips Symposium (HCS)*, 2007, pp. 1–10.

[4] S. H. S. Ariffin, J. A. Schormans, and A. Ma, "Application of the generalised ballot theorem for evaluation of performance in packet buffers with non-first in first out scheduling," *IET Communications*, vol. 3, no. 6, pp. 933–944, 2009.

[5] Barefoot, "The World's Fastest & Most Programmable Networks," *White paper*, 2018. [online] Available: `https://www.barefootnetworks.com/resources/worlds-fastest-most-programmable-networks/`

[6] N. Bastin and R. McGeer, "Programmable, Controllable Networks," in *The GENI Book*. Springer, 2016, pp. 149–178.

[7] I. Benacer, F.-R. Boyer, and Y. Savaria, "A High-Speed Traffic Manager Architecture for Flow-Based Networking," in *IEEE International New Circuits and Systems Conference (NEWCAS)*, June 2017, pp. 161–164.

[8] I. Benacer, F.-R. Boyer, and Y. Savaria, "Design of a Low Latency 40 Gb/s Flow-Based Traffic Manager Using High-Level Synthesis," in *IEEE International Symposium on Circuits and Systems (ISCAS)*, May 2018, pp. 1–5.

[9] I. Benacer, F.-R. Boyer, and Y. Savaria, "A Fast, Single-Instruction–Multiple-Data, Scalable Priority Queue," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 26, no. 10, pp. 1939–1952, Oct. 2018.

[10] I. Benacer, F.-R. Boyer, and Y. Savaria, "HPQS: A Fast, High-Capacity, Hybrid Priority Queuing System for High-Speed Networking Devices," *submitted to the IEEE Access*, 2019.

[11] I. Benacer, F.-R. Boyer, and Y. Savaria, "A High-Speed, Scalable, and Programmable Traffic Manager Architecture for Flow-Based Networking," *IEEE Access*, vol. 7, pp. 2231–2243, 2019.

[12] I. Benacer, F.-R. Boyer, N. Bélanger, and Y. Savaria, "A Fast Systolic Priority Queue Architecture for a Flow-Based Traffic Manager," in *IEEE International New Circuits and Systems Conference (NEWCAS)*, 2016, pp. 1–4.

[13] I. Benacer, F.-R. Boyer, and Y. Savaria, "HPQ: A High Capacity Hybrid Priority Queue Architecture for High-Speed Network Switches," in *IEEE International New Circuits and Systems Conference (NEWCAS)*, June 2018, pp. 229–233.

[14] J. C. Bennett and H. Zhang, "Hierarchical packet fair queueing algorithms," *IEEE/ACM Transactions on Networking (ToN)*, vol. 5, no. 5, pp. 675–689, 1997.

[15] R. Bhagwan and B. Lin, "Fast and scalable priority queue architecture for high-speed network switches," in *Proceedings of the Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM)*, vol. 2, 2000, pp. 538–547.

[16] G. Bloom, G. Parmer, B. Narahari, and R. Simha, "Shared hardware data structures for hard real-time systems," in *Proceedings of the ACM International Conference on Embedded Software*, series EMSOFT '12. ACM, 2012, pp. 133–142.

[17] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese *et al.*, "P4: Programming Protocol-independent Packet Processors," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 3, pp. 87–95, 2014.

[18] W. Braun and M. Menth, "Software-Defined Networking Using OpenFlow: Protocols, Applications and Architectural Design Choices," *Future Internet*, vol. 6, no. 2, pp. 302–336, 2014.

[19] G. Brebner, "Extending the range of P4 programmability," *Keynote in the First European P4 workshop (P4EU)*, Sep. 2018.

[20] Broadcom, "Integrated Switch and Traffic Manager," *TME-2000 Product Brief*, 2008.

[21] Broadcom, "200G Integrated Packet Processor, Traffic Manager, and Fabric Interface Single-Chip Device," *BCM88650 series world most dense 100 GbE switching solution*, 2012.

[22] Broadcom, "High-Capacity StrataXGS Trident II Ethernet Switch Series," *BCM56850 Series*, 2018. [online] Available: `https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs/bcm56850-series`

[23] J. Brown, S. Woodward, B. Bass, and C. Johnson, "IBM Power Edge of Network Processor: A Wire-Speed System on a Chip," *IEEE Micro*, vol. 31, no. 2, pp. 76–85, 2011.

[24] R. Brown, "Calendar Queues: A Fast O(1) Priority Queue Implementation for the Simulation Event Set Problem," *Communications of the ACM*, vol. 31, no. 10, pp. 1220–1227, 1988.

[25] CAIDA, "CAIDA Data — Overview of Datasets Monitors and Reports," *Center for Applied Internet Data Analysis*, 2017. [online] Available: `http://www.caida.org/data/passive/`

[26] D. Calum and W. Dan, "Understanding 5G: Perspectives on future technological advancements in mobile," *Analysis report*, 2014.

[27] R. Chandra and O. Sinnen, "Improving application performance with hardware data structures," in *IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW)*, 2010, pp. 1–4.

[28] F. M. Chiussi, A. Brizio, A. Francini, K. Grant, K. Kazi, D. Khotimsky, S. Krishnan, S. Shen, M. Syed, and T. Wasilewski, "A Family of ASIC devices for Next Generation Distributed Packet Switches with QoS support for IP and ATM," in *IEEE Hot Interconnects*, 2001, pp. 145–149.

[29] L. H. Crockett, R. A. Elliot, M. A. Enderwitz, and R. W. Stewart, *The Zynq Book: Embedded Processing with the ARM Cortex-A9 on the Xilinx Zynq-7000 All Programmable SoC*. Strathclyde Academic Media, 2014.

[30] J. S. da Silva, F.-R. Boyer, L.-O. Chiquette, and J. P. Langlois, "Extern Objects in P4: an ROHC Header Compression Scheme Case Study," in *IEEE Conference on Network Softwarization and Workshops (NetSoft)*, 2018, pp. 517–522.

[31] EZchip, "NP-5 240 Gbps NPU for Carrier Ethernet applications," *Product Brief*, 2015.

[32] EZchip, "NPS-400 400 Gbps NPU for Smart Networks," *Product Brief*, 2015.

[33] H. Fallside, "Queue Manager Reference Design," *Xilinx Application Note XAPP511, Xilinx, Inc.*, 2007.

[34] F. F. Fariborz and M. Ait Otmane, "A New 10 Gbps Traffic Management Algorithm for High-Speed Networks," in *IEEE International Symposium on Circuits and Systems (ISCAS)*, 2007, pp. 2510–2513.

[35] M. A. Franklin, P. Crowley, H. Hadimioglu, and P. Z. Onufryk, *Network Processor Design: Issues and Practices*. Elsevier, 2003, vol. 2.

[36] R. Giladi, *Network Processors: Architecture, Programming, and Implementation*. Morgan Kaufmann, 2008.

[37] A. Gupta and R. K. Jha, "A Survey of 5G Network: Architecture and Emerging Technologies," *IEEE Access*, vol. 3, pp. 1206–1232, 2015.

[38] P. K. Gupta, "Xeon+FPGA Platform for the Data Center," in *Workshop on the Intersections of Computer Architecture and Reconfigurable Logic*, vol. 119, 2015.

[39] M. Huang, K. Lim, and J. Cong, "A scalable, high-performance customized priority queue," in *IEEE International Conference on Field Programmable Logic and Applications (FPL)*, 2014, pp. 1–4.

[40] Intel, "Enabling Quality of Service With Customizable Traffic Managers," *White paper*, 2005.

[41] A. Ioannou and M. G. Katevenis, "Pipelined Heap (Priority Queue) Management for Advanced Scheduling in High-Speed Networks," *IEEE/ACM Transactions on Networking (ToN)*, vol. 15, no. 2, pp. 450–461, 2007.

[42] A. Khan, K. Patel, A. Aurora, A. Raza, B. Parruck, A. Bagchi, A. Ghosh, B. Litinsky, E. Hong, E. Zhao *et al.*, "Design and development of the first single-chip full-duplex OC48 traffic manager and ATM SAR SoC," in *Proceedings of the IEEE Custom Integrated Circuits Conference*, 2003, pp. 35–38.

[43] D. E. Knuth, *The Art of Computer Programming.* Addison-Wesley, 1998, vol. 3.

[44] R. Krishnamurthy, S. Yalamanchili, K. Schwan, and R. West, "ShareStreams: A Scalable Architecture and Hardware Support for High-Speed QoS Packet Schedulers," in *IEEE Annual Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2004, pp. 115–124.

[45] N. C. Kumar, S. Vyas, J. A. Shidal, R. Cytron, C. Gill, J. Zambreno, and P. H. Jones, "Improving System Predictability and Performance via Hardware Accelerated Data Structures," *Procedia Computer Science*, vol. 9, pp. 1197–1205, 2012.

[46] N. C. Kumar, S. Vyas, R. K. Cytron, C. D. Gill, J. Zambreno, and P. H. Jones, "Hardware-software architecture for priority queue management in real-time and embedded systems," *International Journal of Embedded Systems*, vol. 6, no. 4, pp. 319–334, 2014.

[47] A. Lara, A. Kolasani, and B. Ramamurthy, "Network Innovation using OpenFlow: A Survey," *IEEE Communications Surveys Tutorials*, vol. 16, no. 1, pp. 493–512, First Quarter 2014.

[48] P. Lavoie, D. Haccoun, and Y. Savaria, "A systolic architecture for fast stack sequential decoders," *IEEE Transactions on Communications*, vol. 42, no. 234, pp. 324–335, 1994.

[49] T. Leighton, "Tight Bounds on the Complexity of Parallel Sorting," *IEEE Transactions on Computers*, vol. C-34, no. 4, pp. 344–354, April 1985.

[50] R. Mari, "In-Band Network Telemetry - A Powerful Analytics Framework for your Data Center," *Open Compute Project (OCP) summit*, Mar. 2018.

[51] Marvell, "Marvell Introduces New Xelerated Network Processors and Traffic Management Solutions for the Mobile Internet," *News Release*, 2013. [online] Available: `https://www.marvell.com/`

[52] A. Mathur, M. Fujita, E. Clarke, and P. Urard, "Functional Equivalence Verification Tools in High-Level Synthesis Flows," *IEEE Design & Test of Computers*, vol. 26, no. 4, pp. 88–95, 2009.

[53] K. McLaughlin, S. Sezer, H. Blume, X. Yang, F. Kupzog, and T. Noll, "A Scalable Packet Sorting Circuit for High-Speed WFQ Packet Scheduling," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 16, no. 7, pp. 781–791, 2008.

[54] K. McLaughlin, D. Burns, C. Toal, C. McKillen, and S. Sezer, "Fully hardware based WFQ architecture for high-speed QoS packet scheduling," *Integration, the VLSI journal*, vol. 45, no. 1, pp. 99–109, 2012.

[55] Mentor, "Catapult® High-Level Synthesis," *High-Level Synthesis and RTL Low-Power*, 2018. [online] Available: `https://www.mentor.com/hls-lp/catapult-high-level-synthesis/`

[56] A. Mihal, C. Kulkarni, M. Moskewicz, M. Tsai, N. Shah, S. Weber, Y. Jin, K. Keutzer, K. Vissers, C. Sauer *et al.*, "Developing architectural platforms: A disciplined approach," *IEEE Design & Test of Computers*, vol. 19, no. 6, pp. 6–16, 2002.

[57] S.-W. Moon, J. Rexford, and K. G. Shin, "Scalable hardware priority queue architectures for high-speed packet switches," *IEEE Transactions on Computers*, vol. 49, no. 11, pp. 1215–1227, 2000.

[58] C. Ni, C. Gan, and H. Chen, "Joint bandwidth allocation on dedicated and shared wavelengths for QoS support in multi-wavelength optical access network," *IET Communications*, vol. 7, no. 16, pp. 1863–1870, 2013.

[59] A. Nikologiannis, I. Papaefstathiou, G. Kornaros, and C. Kachris, "An FPGA-based queue management system for high speed networking devices," *Microprocessors and Microsystems*, vol. 28, no. 5, pp. 223 – 236, 2004, Special Issue on FPGAs: Applications and Designs.

[60] S. O'Neill, R. F. Woods, A. J. Marshall, and Q. Zhang, "A Scalable and Programmable Modular Traffic Manager Architecture," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 4, no. 2, p. 14, 2011.

[61] P4 Language Consortium, "P4 language specification," *version 1.1.0*, 2018. [online] Available: `https://p4.org/specs/`

[62] N. Panwar, S. Sharma, and A. K. Singh, "A survey on 5G: The next generation of mobile communication," *Physical Communication*, vol. 18, pp. 64–84, 2016.

[63] I. Parvez, A. Rahmati, I. Guvenc, A. I. Sarwat, and H. Dai, "A Survey on Low Latency Towards 5G: RAN, Core Network and Caching Solutions," *IEEE Communications Surveys Tutorials*, vol. 20, no. 4, pp. 3098–3130, Fourthquarter 2018.

[64] P. G. Paulin, C. Pilkington, M. Langevin, E. Bensoudane, D. Lyonnard, O. Benny, B. Lavigueur, D. Lo, G. Beltrame, V. Gagné, and G. Nicolescu, "Parallel programming models for a multiprocessor SoC platform applied to networking and multimedia," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 14, no. 7, July 2006.

[65] J. Peisa, "5G Techniques for Ultra Reliable Low Latency Communication," *Keynote in IEEE Conference on Standards for Communications & Networking*, 2017. [online] Available: `http://cscn2017.ieee-cscn.org/files/2017/08/Janne_Peisa_Ericsson_CSCN2017.pdf`

[66] H. W. Poole, L. Lambert, C. Woodford, and C. J. Moschovitis, *The Internet: A Historical Encyclopedia*. Abc-Clio Inc., 2005, vol. 2.

[67] G. Reinman and N. P. Jouppi, "CACTI 2.0: An integrated cache timing and power model," *Western Research Lab Research Report*, vol. 7, 2000.

[68] R. Rönngren and R. Ayani, "A comparative study of parallel and sequential priority queue algorithms," *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, vol. 7, no. 2, pp. 157–209, 1997.

[69] N. Shah and K. Keutzer, "Network Processors: Origin of Species," in *17th International Symposium of Computer and Information Science*, 2002.

[70] M. Shreedhar and G. Varghese, "Efficient fair queuing using deficit round-robin," *IEEE/ACM Transactions on Networking (ToN)*, vol. 4, no. 3, pp. 375–385, June 1996.

[71] A. Sivaraman, A. Cheung, M. Budiu, C. Kim, M. Alizadeh, H. Balakrishnan, G. Varghese, N. McKeown, and S. Licking, "Packet Transactions: High-Level Programming for Line-Rate Switches," in *Proceedings of the 2016 ACM SIGCOMM Conference*, 2016, pp. 15–28.

[72] A. Sivaraman, S. Subramanian, M. Alizadeh, S. Chole, S.-T. Chuang, A. Agrawal, H. Balakrishnan, T. Edsall, S. Katti, and N. McKeown, "Programmable Packet Scheduling at Line Rate," in *Proceedings of the ACM SIGCOMM Conference*, 2016, pp. 44–57.

[73] S. Stanley, "Traffic Managers Update," *LightReading Networking the Communications Industry*, 2004. [online] Available: `https://www.lightreading.com/`

[74] Synopsys, "Synopsys Introduces Synphony High Level Synthesis," *Press Releases*, 2017. [online] Available: `https://news.synopsys.com/index.php?s=20295&item=123096`

[75] T. N. Van, V. T. Thien, S. N. Kim, N. P. Ngoc, and T. N. Huu, "A high throughput pipelined hardware architecture for tag sorting in packet fair queuing schedulers," in *IEEE International Conference on Communications, Management and Telecommunications (ComManTel)*, 2015, pp. 41–45.

[76] H. Wang and B. Lin, "Pipelined van Emde Boas Tree: Algorithms, analysis, and applications," in *IEEE International Conference on Computer Communications (INFOCOM)*, 2007, pp. 2471–2475.

[77] H. Wang and B. Lin, "Succinct priority indexing structures for the management of large priority queues," in *17th IEEE International Workshop on Quality of Service (IWQoS)*, 2009, pp. 1–5.

[78] H. Wang and B. Lin, "Per-Flow Queue Management with Succinct Priority Indexing Structures for High Speed Packet Scheduling," *IEEE Transactions on Parallel and Distributed Systems*, vol. 24, no. 7, pp. 1380–1389, 2013.

[79] Xilinx, Inc., "Traffic Management in Xilinx FPGAs," *White paper, ver. 1.0, WP244*, 2006.

[80] Xilinx, Inc., "ML631 Virtex-6 HXT FPGA Packet Processor/Traffic Manager Evaluation Board User guide," *UG841 ver. 1.0*, 2012.

[81] Xilinx, Inc., "Vivado Design Suite AXI Reference Guide," *UG1037, v4.0*, 2017.

[82] Xilinx, Inc., "Vivado High-Level Synthesis," *Accelerates IP Creation by Enabling C, C++ and System C Specifications*, 2018. [online] Available: `https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html`

[83] Xilinx, Inc., *Vivado Design Suite User Guide–High-Level Synthesis.* Xilinx, 2018, UG902, v2018.1.

[84] Y. Xu, K. Li, J. Hu, and K. Li, "A genetic algorithm for task scheduling on heterogeneous computing systems using multiple priority queues," *Information Sciences*, vol. 270, pp. 255–287, 2014.

[85] M. Yu, J. Rexford, M. J. Freedman, and J. Wang, "Scalable flow-based networking with DIFANE," *ACM SIGCOMM Computer Communication Review*, vol. 41, no. 4, pp. 351–362, 2011.

[86] Q. Zhang, R. Woods, and A. Marshall, "An On-Demand Queue Management Architecture for a Programmable Traffic Manager," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, no. 10, pp. 1849–1862, 2012.

[87] X. Zhuang and S. Pande, "A Scalable Priority Queue Architecture for High Speed Network Processing," in *Proceedings of the IEEE International Conference on Computer Communications (INFOCOM)*, 2006, pp. 1–12.

APPENDIX A SUMMARY OF NOTATION

$\{\,\}$ Unordered set of elements.

$\langle\,\rangle$ Ordered vector of elements.

$\langle\{\,\}\rangle$ Elements of the set can be placed in any order in the vector.

$A \setminus B$ Set difference.

$\{\{X\}\} = \{X\}$, i.e., sets are flattened.

2nd max X = max $(X \setminus \{\max X\})$, i.e., the second largest element.